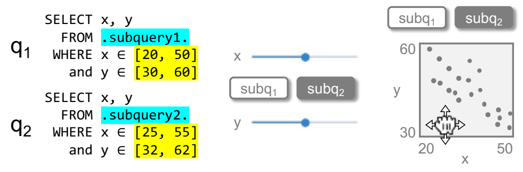
PI$_2$: Generating Visual Analysis Interfaces From Queries

Yiru Chen
Columbia University
yiru.chen@columbia.edu

Eugene Wu
DSI, Columbia University
ewu@cs.columbia.edu

ABSTRACT

Interactive visual analysis interfaces are critical in nearly every data task. Yet creating new interfaces is deeply challenging, as it requires the developer to understand the queries needed to express the desired analysis task, design the appropriate interface to express those queries for the task, and implement the interface using a combination of visualization, browser, server, and database technologies. Although prior work generates a set of interactive widgets that can express an input query log, this paper presents PI$_2$, the first system to generate fully functional visual analysis interfaces from an example sequence of analysis queries. PI$_2$ analyzes queries syntactically and represents a set of queries using a novel DIFFTREE structure that encodes systematic variations between query abstract syntax trees. PI$_2$ then maps each DIFFTREE to a visualization that renders its results, the variations in each DIFFTREE to interactions, and generates a good layout for the interface. We show that PI$_2$ can express data-oriented interactions in existing visualization interaction taxonomies, can reproduce or improve several real-world visual analysis interfaces, generates interfaces in 2 – 19s (median 6s), and scales linearly with the number of queries.



q$_1$
```
SELECT x, y
  FROM .subquery1.
 WHERE x ∈ [20, 50]
   and y ∈ [30, 60]
```
q$_2$
```
SELECT x, y
  FROM .subquery2.
 WHERE x ∈ [25, 55]
   and y ∈ [32, 62]
```

(a) Input queries (b) PI$_1$ output (c) This paper (PI$_2$)

Figure 1: Comparison between prior and this work. (b) Prior work generates an unordered set of interaction widgets that express the two queries. (c) This paper presents a novel model that accounts for widgets, layouts, and interactive visualizations, and generate fully interactive interfaces.

1 INTRODUCTION

Interactive visual analysis interfaces (or simply *interfaces*) are critical in nearly every stage of data management, including data cleaning [54], wrangling [23], modeling [16], exploration [35], and communication [12, 17]. Interfaces empower the user to easily express relevant analysis queries using interactive controls that hide the underlying query complexity. A prominent example is VizQL [47] (commercialized as Tableau), which was carefully designed for analyses based on OLAP cube queries. However, analyses are not restricted to OLAP queries and can be arbitrarily complex. Thus, translating a custom analysis query workload into a fully functional interface remains deeply challenging.

EXAMPLE 1. *The two queries in Figure 1(a) differ in two ways: the from clause chooses from two subqueries, and the two predicate ranges may change. Designing a good interface requires numerous nuanced decisions. Should the queries be rendered together or separately? As scatterplots (supports panning along the y-axis), bar charts (which do not), or another chart type? How should the user choose the subquery? Should textboxes, sliders, range sliders, or panning specify the range predicates? Should the layout be horizontal or vertical? What if the screen is wide? Or narrow? Figure 1(c) depicts a sensible interface that expresses these queries. It visualizes the query results in a scatter plot. The user can drag the plot to change the x and y predicate ranges, and click on the buttons to select the desired subquery.*

Once a design is established, the developer still needs to ensure that interactions appropriately transform the underlying queries; implement the interface using a mix of browser, visualization, server, and database technologies; debug her implementation; and finally deploy the result. Only then, can she finally use or share the interface.

The example highlights how intimidating and laborious it is to navigate design decisions and build interfaces using a multitude of technologies. To this end, dashboard creators (e.g., Metabase [31], Retool [40]) and exploration tools (e.g., Tableau [35]) help author SQL-based analysis dashboards, however they limit the scope of analyses and queries in order to keep their own interfaces simple. For instance, Metabase is restricted to parameterized queries [32], and Tableau to OLAP cube queries [50]. Analyses that go beyond these restrictions will require manual implementation.

To simplify interface implementation, numerous programming libraries have been created to recommend [52, 58] or create [5, 41] visualizations, manipulate the DOM [21], manage a web server [13], and construct SQL queries [25]. Similarly, tools such as AirTable [2], Figma [11], and Plasmic [37] focus on rapid interface design. Ultimately, each only solves one step, and programming expertise and effort is needed to use and combine these tools.

An ideal system would directly generate an interface using example analysis queries. Prior work [57] (PI$_1$) was a first step in this direction. It modeled interfaces as a visualization that renders and underlying query's result, and widgets that syntactically transform the underlying query; the widgets defined the set of all queries that the interface can express. PI$_1$ modeled each query as an abstract syntax tree (AST), aligned the trees, and extracted the subtrees that differ. It then grouped those differences and mapped each group to an interactive widget. In this way, it returned a set of widgets that expresses the input set of queries (and perhaps other queries).

Unfortunately, PI$_1$ has several fundamental drawbacks. Its output is limited to unordered set of widgets (e.g., Figure 1(b)) because it doesn't consider how the query results are rendered. Thus, it cannot support interactions within visualizations (e.g., pan&zoom, selection) nor multiple visualizations in the same interface. Further, it generates a flat mapping from syntactic differences to interactive widgets, however the flat mapping cannot model hierarchical interface layouts nor nested widgets (such as tabs containing widgets).

This paper presents PI_2, the first system to generate interactive multi-visualization interfaces (such as Figure 1(c)) from a small number of example queries. To do so, we propose a new interface generation model that is based on schema matching. We first extend abstract syntax trees with four types of *choice nodes* to encode subtree variations between queries. Choice nodes directly correspond to grammar production rules. For instance, the ANY choice node is used to choose one of its children—such as the two subqueries in Figure 1(a))—and corresponds to an ordered choice production rule.

Each tree represents a subset of input queries, and we map the set of trees to an interface. Specifically, each tree's result table is mapped to a visualization, each choice node is mapped to a widget or visualization interaction, and the tree structure is mapped to a hierarchical layout. By defining transform rules that merge, combine, and transform these trees, we are able to search the space of tree structures that result in different candidate interface designs. Finally, we rank candidate interfaces by combining existing interface cost models [15, 27, 57] to estimate how easily the user can use the interface to express the sequence of input queries. PI_2 only requires access to a lightly annotated language grammar and database catalogue in order to determine valid mappings.

Informally, our technical problem is: given an input sequence of queries, search the space of extended ASTs and interface mappings to identify the lowest cost interface. We use Monte Carlo Tree Search [10, 42] (MCTS) to balance exploration of diverse tree structures with exploitation of good tree structures found so far, and generate complex multi-view interfaces in seconds. We contribute:

- PI_2, the first system to generate fully functional multi-view interfaces from example analysis queries. The system is database agnostic, and only needs access to the query grammar, a database connection to execute queries, and the database catalogue.
- A novel model that unifies SQL query strings, interactive visualizations, widgets, and interface layout. The model enables us to formulate interface mapping in terms of schema mapping.
- An evaluation that shows PI_2 expresses all data-related interactions in Yi et al.'s [55] visualization interaction taxonomy. PI_2 shows that small difference in analysis queries can considerably change the interface design, and illustrates the importance of an automated interface generation tool. We further show 3 case studies that use real-world queries to improve the SDSS web search interface, reproduce Google's Covid-19 visualization, and show how to use queries to author a sales analysis dashboard.
- A set of simple optimizations that reduce interface generation times from 30s to a median of 6s. We further find that PI_2 runtime scales linearly with the number of input queries.

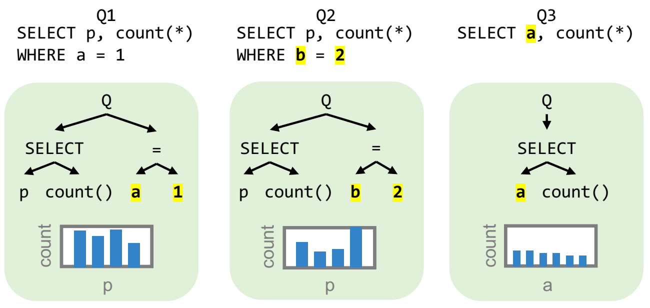
Scope of this work: PI_2 is designed to generate task-specific analysis interfaces. It assumes a small sequence of coherent queries that represent a desired analysis, and is not suitable for open-ended exploration queries that are often unrelated and seemingly random. Further, PI_2 is the first to show that end-to-end interface generation from queries *is even possible*, and we leave scalability and customizability of the generated interfaces to future work.

2 INTERFACE GENERATION OVERVIEW

PI_2 transforms an input sequence of queries into an interactive interface in four steps: parsing queries into a generalization of abstract syntax trees (ASTs) that we call DiffTrees, mapping the DiffTrees to a candidate interface, estimating the interface's cost, and either returning the interface or transforming the DiffTrees to generate a new candidate interface. This section walks through these steps and introduces key concepts using a simple example.

Static Interfaces: Figure 2 lists three input queries on table T where attribute p, a, b are all integers. Q1 and Q2 transform the predicate attribute and literal, and Q3 selects a instead of p. PI_2 first parses each query into their corresponding DiffTree (simply a normal AST). Each DiffTree is rendered as a visualization; since the DiffTrees are static, the interface for these three queries consists of three static charts. For brevity, we omit the FROM and GROUPBY clauses and show simplified syntax trees.

Interactive Interfaces: Let us temporarily focus on the differing predicate in Q1 and Q2 to show how different DiffTrees structures can result in different interface designs. For instance, Figure 3(a) is rooted at an ANY node whose children are the two predicates. ANY is a *Choice Node* that can choose one of its child subtrees. In general,



```
Q1
SELECT p, count(*)
WHERE a = 1
```

```
Q2
SELECT p, count(*)
WHERE b = 2
```

```
Q3
SELECT a, count(*)
```

Figure 2: Example of three queries and their simplified ASTs. A static interface would render one chart for each query.

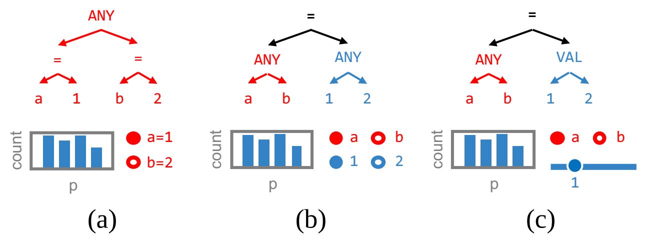


Figure 3: Three examples of DiffTrees for Q1, Q2, focusing on the subtree for the predicate. The ANY choice node can choose one of its children. (a) Each predicate can be chosen using a radio button that parameterizes the ANY node, (b) the left and right operands can be individually chosen using radio buttons, (c) the literal operand is generalized beyond the input values 1 and 2.

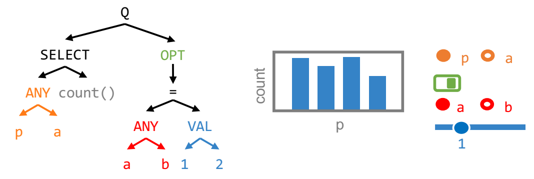


Figure 4: A DIFFTREE for Q1-3 and a candidate interface.

choice nodes encode subtree variations[1] that the user can control through the interface. In the example, the ANY node is mapped to two radio buttons (other widgets such as a dropdown are valid as well), where clicking on the first button would bind the ANY to its first child a=1. The DIFFTREE output is visualized as a bar chart.

Tree Transformations: Note that both of ANY's children are rooted at =, and can thus be pushed above the ANY node. This is an example of a *Tree Transformation Rule* that we describe in Section 6. The resulting DIFFTREE in Figure 3(b) shows two ANY nodes that can independently choose the left and right operands. This leads to an interface with two interactions (radio buttons), and also generalizes the interface beyond the input queries. For instance, the query can now express `SELECT p, count(*) WHERE b=1`.

Schemas: A single DIFFTREE can be mapped to many interface designs, each with different widgets, visualization interactions, and layouts. For instance, although Figure 3(b) treated the second ANY's children 1 and 2 as generic subtrees, we can easily infer that they are numerically typed. Further, the equality comparison tells us their values are defined by the domains of attributes a and b. Thus, with access to the database catalogue, we can infer that the second ANY node's *Schema* is the union type of a and b, which are both numeric. We can then generalize the ANY to a VAL choice node that replaces itself with the literal that it is bound to (Figure 3(c)). For instance, when the user changes the slider position to 5, the value in bound to the VAL node, which resolves itself to 5. This lets us map the ANY node to a numeric slider that is initialized with the minimum and maximum of attribute a and b's domains. This was an example of generalization based on relaxing a choice node's schema (Section 3.2), and a tree transformation rule that generalizes the ANY node's schema and replaces it with the VAL node.

All Three Queries: Now, let us add Q3. The simplest interface would be to partition the queries into two clusters, where Q3 is rendered as a static chart, and Q1 and Q2 is mapped to one of the interactive interfaces discussed so far. We can then choose to lay them out horizontally or vertically (Section 4.3). Another possibility is to merge all three queries into a single DIFFTREE, which would map to an interface with a single visualization. Figure 4 illustrates one possible DIFFTREE structure, where an ANY node in the SELECT clause chooses to project p or a. This maps to an interface similar to Figure 3(c), but with a radio button to choose the attribute to project and another toggle button to express optional status of the where clause. Naturally, which of these possible interface designs (or others not discussed here) that should be generated and returned to the user depends on many factors, such as usability, layout, accessibility, and other factors that are difficult to quantify.

[1]Choice nodes can be viewed as generalizing parameterized expressions in SQL to parameterizing arbitrary syntax structures in a query.

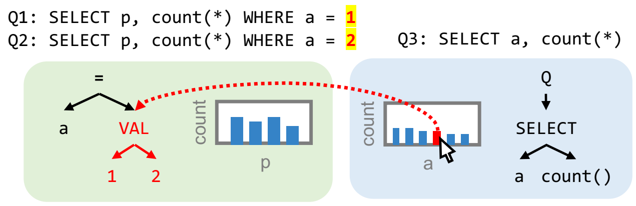

Q1: SELECT p, count(*) WHERE a = 1
Q2: SELECT p, count(*) WHERE a = 2
Q3: SELECT a, count(*)
=
a
VAL
1 2
count
p
count
a
Q
SELECT
a count()

Figure 5: Multi-view interface where clicking on the right-side chart updates the left chart.

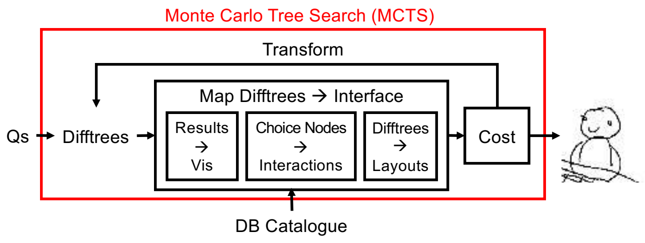

Monte Carlo Tree Search (MCTS)
Transform
Qs
Difftrees
Map Difftrees → Interface
Results → Vis
Choice Nodes → Interactions
Difftrees → Layouts
Cost
DB Catalogue

Figure 6: PI$_2$ interface generation pipeline.

Quantitative interface evaluation is an active area of research, and Section 5 presents the best practices used to develop this paper's cost function and its limitations.

Multi-view Interfaces PI$_2$ can also generate interactive multi-view interfaces. Figure 5 illustrates a slightly different set of queries, where the Q1 and Q2 only differ in the literal, and Q3 remains the same. Since the literal is compared to attribute a, an alternative to mapping the VAL node to a slider is to map it to a *visualization interaction* in Q3's bar chart. Specifically, each bar is derived from a and count(*) in Q3's result. Thus, clicking on a bar can also derive a valid value in attribute a's domain that can bind to the VAL node.

Summary and Generation Pipeline: To summarize, PI$_2$ generates interfaces in a four-step process. We first parse the input query sequence Q into DIFFTREES, and map the DIFFTREES into an interface. An interface mapping $\mathbb{I} = (\mathbb{V}, \mathbb{M}, \mathbb{L})$ is defined by mapping each DIFFTREE result to a visualization ($\mathbb{V}$), choice nodes to interactions (widget or visualization interaction) ($\mathbb{M}$), and a layout tree ($\mathbb{L}$). A cost function $C(\mathbb{I}, \mathbb{Q})$ evaluates the interface and either returns the interface or choose a valid transformation to apply to the DIFFTREES. Informally, given queries Q and cost function C, our problem is to return the lowest cost interface $\mathbb{I}$ that can express all queries in the log. The formal statement is presented in Section 6.

We solve this problem using Monte Carlo Tree Search [6] (MCTS), a search algorithm for learning good game-playing strategies and famously used in AlphaGo [46]. It balances exploitation of good explored states (DIFFTREE structures), and exploration of new states. We describe our search procedure and optimizations in Section 6.2.

3 DIFFTREES

DIFFTREES extend ASTs with *Choice Nodes* that encode the structural differences between the queries, and are the bridge from input queries to the output interface. Specifically, an interface $\mathbb{I} = (\mathbb{V}, \mathbb{M}, \mathbb{L})$ is defined by mapping each DIFFTREE result to a visualization ($\mathbb{V}$), choice nodes to interactions ($\mathbb{M}$), and a layout tree ($\mathbb{L}$). Candidate visualization and interaction mappings are based on schema matching between the Difftree result and visualization schemas, and choice node and interaction schemas respectively. This section defines Difftrees and how their result and choice node schemas are inferred. The next section will present visualization and interaction schemas and the formal mapping procedure.

3.1 Difftree and Choice Nodes

A Difftree Δ compactly represents a set of expressible ASTs $\{\Delta\} = \{\delta_1, \delta_2, ...\}$, where δ_i is an AST. It extends ASTs with *Choice Nodes*–`ANY`, `VAL`, `MULTI`, and `SUBSET`–that correspond to production rules in a PEG grammar. This supports arbitrarily complex subtrees, yet can still be analyzed because the set of variations is predefined and finite. Finally, Difftrees guarantee that any expressible AST is syntactically correct. We describe Difftree node types below, the set of ASTs they express, and how each choice node resolves to an AST subtree when bound to a set of parameters.

- **`ANY(c1,..,ck)`** can choose one of its k children, akin to the production rule `ANY`$\rightarrow$`c1|..|ck`. When bound to an index $i \in [1, k]$, it resolves to c_i. For instance, binding 1 to the `ANY` in Figure 3(a) will resolve it to `a=1`. A special case is when `ANY` has two children, where one is an empty subtree. We call this `OPT` for optional, and is useful for mapping to binary interactions such as toggles. This node expresses the ASTs: $\cup_{i \in [1,k]}\{c_i\}$.
- **`VAL(c1,..,ck)`** represents a literal that matches a regex pattern in a grammar. Its children are all literals and its value domain is defined by the union of its children's types. In practice, it is a pass-through node that resolves to any value it is bound to. For instance, the `VAL` in Figure 3(c) resolves to the slider's value. The next subsection describes types in more detail. Let $c_i.d$ refer to a child node's domain, then this node expresses $\cup_{i \in [1,k]} c_i.d$.
- **`MULTI[sep](c)`** represents lists that express e.g., project lists, group-by lists, and conjunctions. It expresses the production rule `MULTI`$\rightarrow$`c (sep c)*`. It repeats its child `c` one or more times, where its child may also be a Difftree. When it is bound to a list of parameterizations $[p_1, .., p_k]$, it passes each parameterization p_i to its child and concatenates their resolved ASTs using `sep`. This node expresses the ASTs: $\{sep\,(t)\,|t \in \{c\}^k \wedge k \in \mathbb{N}_0\}$ that correspond to an arbitrary number of cross products between all elements of $\{c\}$.
- **`SUBSET[sep](c1,..,ck)`** represents the production rule `SUBSET`$\rightarrow$ `c1?..ck?` with `sep` as the separator. Bind a set of indices between 1 and k to it will resolve to the corresponding subset of its children, concatenated with the separator `sep`. The node expresses the ASTs: $\{sep\,(t)\,|t \subseteq \{c1, c2, ...ck\}\}$ that corresponds to all possible subsets of the children.
- **Non-choice Nodes:** Finally, let `N(c1,..,ck)` be a non-choice node. If N is a leaf node, it expresses the ASTs: $\{N\}$. Otherwise, it expresses $\{N(t)|t \in \{c_1\} \times \ldots \times \{c_k\}\}$. Let $c_{s_1}, \ldots, c_{s_n}$ be the subset of children whose subtrees contain one or more choice nodes. When N is bound to a list of parameterizations $[p_1, \ldots, p_n]$, it passes each p_i to child c_{s_i}.

3.2 Schemas

Although PI_2 does not reproduce the database front-end's type checking, it infers type and schema information to map Difftrees

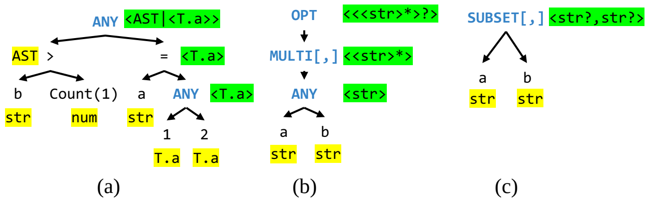


Figure 7: Example Difftrees of the four types of choice nodes, with type and schema annotations.

to an interface. Choice nodes and their ancestors are annotated with schemas, and all other nodes are annotated with types. We distinguish these because only choice nodes (or their ancestors) are mapped to interactions. We use *Result Schema* to refer to the schema of the Difftree's result, and *Node Schema* to refer to the schema of a Difftree node.

3.2.1 *Types.*

All non-choice nodes that are not ancestors of a choice node are annotated with type information. A type defines a domain of values [1] that a node can express. For simplicity we describe a trivial type hierarchy of primitive types: AST$\rightarrow$str$\rightarrow$num. num specializes str, and str specializes AST; AST expresses any abstract syntax tree. Further, each database attribute `a` itself represents an *Attribute Type* and specializes a primitive type's domain to `a`'s domain. In general, internal nodes are of type `AST`, while leaf nodes have more specialized types. We say that a type t_1 is *compatible* with t_2 if its domain is a subset of t_2's domain.

Initialization: We initialize the leaf node's types by using lightweight grammar annotations and the database catalogue. Specifically, we annotate production rules that resolve to str, num with its corresponding type. Also, we infer the type of a function call based on it return type in the catalogue.

Inference: When possible, it is useful to specialize a primitive type to an attribute type. For instance, given $a = 1$, we would like to infer that 1 has type a. To do so, we first lookup attribute names in the catalogue to determine its fully qualified attribute name and domain. We use a simple heuristic based on equality comparison expressions of the form `attr = val`, and assign `val`'s type as `attr`. Finally, we define the union of two types $T_1 \cup T_2$ as their least common ancestor in the type hierarchy. For instance, `str=num`$\cup$`str`, while `num=num`$\cup$`num`.

Example 2 (Node Types). *Figure 7(a) illustrates examples of type inference in* `yellow`. `b`*, and* `a` *are* `str` *types that refer to attribute names (note, they are not attribute types themselves). The catalogue lists* `count()` *as type* `num`. *1, 2 are* `num` *types, however they are compared with* `a` *so their types are specialized to the fully qualified* `T.a`.

3.2.2 *Result Schemas.*

The result schema is defined for a Difftree Δ based on the set of ASTs $\{\delta_1, \delta_2, ...\}$ that it expresses. Specifically, let $s(\delta) =< a_1^\delta : t_1^\delta, .., a_n^\delta : t_n^\delta >$ be the standard result schema of an AST δ without choice nodes, where a_i is the attribute name, and t_i is its type. Δ's result schema is well defined if all $s(\delta \in \{\delta_1, \delta_2, ...\})$ are union compatible. In this case, its result schema is $< a_1 : t_1, \ldots, a_n : t_n >$ where $a_i = \{a_i^\delta | \delta \in \{\delta_1, \delta_2, ...\}\}$, and $t_i = \cup_{\delta \in \{\delta_1, \delta_2, ...\}} t_i^\delta$.

In short, each attribute name is a concatenation of the unique attribute names, and each type is the least compatible type across all expressible ASTs. For example, the result schema in Figure 4 is $< \{T.a \cup T.p\}, num >$. If the schemas are not union compatible, the result schema is undefined.

3.2.3 Node Schemas. A choice node or an ancestor is a *Dynamic Node*, and all other nodes are *Static*. Dynamic nodes are annotated with node schemas that describe the structural variation that they express. Schema $< e_1, \dots, e_n >$ is a list of type expressions, where each expression e_i is a set operators `{|, ?, *}` over types and schemas. `{|, ?, *}` have regular expression semantics where | is the *or* relation (`ANY`), ? is the *existential* relation (`OPT`, `SUBSET`), and $*$ is *repetition* (`MULTI`).

Node Schema Inference: Let $N(c_1, \dots, c_n)$ denote a dynamic node and its n children, where its children may be dynamic or static nodes. Further, let T(N) refer to the node N's type if it is static, and its schema if it is dynamic. We now define schema inference rules that define T(N) for dynamic nodes. Note that schemas may be nested in order to accomodate nested interfaces, such as tabs.

- **`ANY(c1,..,cn)`** considers two conditions. If all children are static, then $T(ANY) =< \cup_{i \in [1,n]} T(c_i) >$. Its schema is the least compatible type of its child types. Otherwise, $< T(c_1) | \dots | T(c_n) >$ is the `OR` of its child schemas.
- **`OPT(c):`** $< T(c)? >$.
- **`MULTI[sep](c):`** $< T(c)* >$, where $*$ denotes 0+ repetitions.
- **`SUBSET[sep](c1,..,cn):`** $< T(c_1)?, \dots, T(c_n)? >$.
- **`AST(c1,..,cn)`** considers two conditions. A static node (Section 3.2.1) has type `AST`. Otherwise, its schema is the cross product of its dynamic children's schemas $< T(c_{s_1}), \dots, T(c_{s_k}) >$, where $c_{s_1}, \dots, c_{s_k}$ are its dynamic children.

EXAMPLE 3. *Figure 7 annotates DIFFTREES with node schemas in* **`green`**. *In Figure 7(a), the bottom* `ANY` *only has static children, so its schema is the union of its child types. The schema of the AST node = is the cross product of its dynamic children's schemas, which is simply* $< T.A >$. *Finally, the top* `ANY` *expresses its left or right child, so has a nested schema* $< AST| < T.a >>$. *Figure 7(b) illustrates nested schemas, where* `MULTI` *applies* `*` *to its child schema, and similarly* `OPT` *applies* ?. *(c) is an example of* `SUBSET`.

3.2.4 Query Bindings. We wish to guarantee that the generated interface can express all of the input queries (and possibly more). Schema information is unfortunately insufficient to ensure this guarantee. For instance, suppose a `VAL` node has type T.a with two child literals *1* and *100* from queries q_1 and q_2. In addition, a bar chart's x-axis renders T.a. Naively, one might expect that clicking on bars in the bar chart can express T.a values, and thus can be bound to the `VAL` node. However, it is possible that the query generating the bar chart filters out all records where T.a = 100, and thus cannot express q_2.

Thus, we also derive the set of *Query Bindings* needed for each dynamic node in order to express all of the input queries. A query binding is a tuple of values consistent with the node's schema. This is done by tracking the binding needed for the DIFFTREES to express each input query and unioning the bindings on a per-node basis.

EXAMPLE 4. *Consider Figure 7(b) and two input queries* `a,a` *and* `b`. *The DIFFTREE bindings are* `{Multi:[{ANY:1}, {ANY:1}]}` *and* `{Multi:[{ANY:2}]}`. *The bindings for each choice node are the union across query bindings. For instance, the bindings for* `MULTI` *are* `{[{ANY:1}, {ANY:1}],[{ANY:2}] }`, *and those for* `ANY` *are* `{1,2}`.

We will refer to Query Bindings when determining whether a candidate interaction mapping is *safe*, described next.

4 INTERFACE MAPPING

PI$_2$ generates candidate visualization ($\mathbb{V}$), interaction ($\mathbb{M}$), and layout ($\mathbb{L}$) mappings in order to generate an interface $\mathbb{I} = (\mathbb{V}, \mathbb{M}, \mathbb{L})$. These candidates define the search space that PI$_2$ explores in Section 6. Visualization and interaction mappings are grounded in schema matching and seeks to ensure safety (that the interface can express all input queries). PI$_2$ is extensible, in that developers can add new visualization types, interaction templates, as well as different types of layouts beyond those used in our prototype. Finally, we will formally present the interface generation problem.

4.1 Visualization Mapping $\mathbb{V}$

$\mathbb{V}$ defines the set of mappings from each DIFFTREE to the visualization that renders its results. Although there are numerous visualization recommendation algorithms, such as ShowMe [29], Draco [34], and Deepeye [26], each is focused on an individual output chart. In contrast, PI$_2$ generates multi-visualization interfaces and needs to take the entire interface into consideration. Specifically, some visualization types, although not individually optimal, may enable visualization interactions that improve the overall interface. For this reason, we use a simple set of heuristics to map a DIFFTREE to a visualization.

Visualizations as Schemas: A visualization renders records from an input table as marks (points, bars) in a chart. Each visualization encodes data attributes using a set of visual variables [3], such as x, y, size, and color, and makes different assumptions about the data types mapped to those visual variables.

As such, we model each visualization type using a *Visualization Schema* $< a_i : t_i, \dots >$, where a_i is the name of a visual variable, and t_i is either Quantitative (Q) or Categorical (C) type. For instance, a bar chart renders categorical values along the x axis, quantitative values along the y axis, and optionally renders categorical values as the bar color.

In addition, a visualization may enforce functional dependency (FD) constraints over the input data. For instance, bar charts assume that x and color functionally determine the y value. This can be inferred if data is a group-by query since the grouping attributes determine the aggregate values, or if the attributes mapped to x and color are unique. Table 1 summarizes the schemas and constraints.

Visualization Mappings: A visualization V can render the result of a DIFFTREE Δ if there is a valid mapping from Δ's result schema S_Δ and the visualization schema S_V such that (1) every data attribute d is mapped to a visual attribute v, (2) each visual attribute is mapped to at most once, (3) every non-optional visual variable is mapped to, and (4) d's type in the result schema is compatible with v's type in the visualization schema. We define compatibility as follows: `str`, and `num` attributes whose cardinality is below 20, are compatible with categorical visual attributes, and `num` attributes are compatible with quantitative visual attributes. Finally,

Vis	Schema and FDs	Interactions
Table	any schema	Click
Point	`<x:Q|C, y:Q,shape:C?, size:C?, color:C?>`	Click, Multi-click, Brush-x/y/xy, Pan, Zoom
Bar	`<x:C, y:Q, color:C?>` (x, color)⟶y	Click, Multi-click Brush-x
Line	`<x:Q|C, y:Q, shape:C?, size:C?, color:C?>` (x, shape, size, color)⟶y	Click, Pan, Zoom

Table 1: Visualization schemas, FD constraints, and supported interactions. `Q` and `C` stand for quantitative (numeric) and categorical types.

we check that the DIFFTREE result satisfies the visualization's constraints based on the query structure (whether it is a group-by) and database schema constraints.

EXAMPLE 5. *Consider* `Q1` *in Figure 2, which groups by* `p` *and computes* `count(*)`*. We can infer that* `p` *determines* `count`*, use the database statistics to estimate that* `p`*'s cardinality is below 20, and thus infer that it can be mapped to a quantitative or categorical visual attribute. The bar chart mapping* $\{p \to x, count \to y\}$ *satisfies the type compatibility and functional dependency constraints, and maps all attributes in the result schema, and has mappings to the required visual attributes.*

Candidate Generation: We generate all valid mappings for a DIFFTREE by iterating through each visualization type, and generating all permutations of the result schema that result in a valid mapping to the visualization schema.

4.2 Interaction Mapping $\mathbb{M}$

$\mathbb{M}$ defines the set of interaction mappings from dynamic nodes to widgets or visualization interactions (collectively called interactions). An interaction mapping $\delta \to I$ from a dynamic node δ to interaction I means that when the user manipulates the interaction, it generates a stream of event tuples whose values bind to δ (Section 3.1 describes node bindings). The goal is to generate $\mathbb{M}$ such that there is a binding for every choice node in the DIFFTREES.

At a high level, PI_2 manages a library of interaction templates, and checks which templates are valid for a given dynamic node. If valid, PI_2 instantiates the interaction with the dynamic node's information, and binds its manipulation event stream to those choice nodes. This subsection first models interactions as schemas and domain constraints, and then defines valid and safe interaction mappings.

4.2.1 Interaction Model. PI_2 manages an extensible library of interaction templates (widgets and visualization interactions). An interaction template defines a schema that is used to identify candidate choice node mappings, and optional constraints that are applied to the choice nodes' query bindings.

An interaction mapping $\delta \to I$ is *valid* if there is a schema match from the dynamic node's schema S_δ to the interaction's schema S_I, and δ's query bindings satisfy the interaction's constraints (if any).

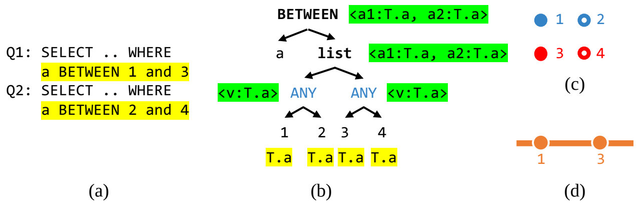


Figure 8: Annotated DIFFTREE for highlighted portion of Q1 and Q2. (b, c) are candidate interaction mappings for the `BETWEEN` or `list` dynamic nodes.

Widgets	Schema	Constraint
Radio, Dropdown, Textbox	`<v:_>`	
Toggle	`<v:_?>`	
Checkbox	`<v:_*>`	
Slider	`<v:num>`	
RangeSlider	`<s:num,e:num>`	s≤e

Table 2: Example widget schemas and constraints. _ matches any schema or type expression.

Specifically, a schema match exists if (1) S_δ and S_I have the same number of type expressions, and (2) each type expression e_i^δ in S_δ is compatible with the corresponding expression e_i^I in S_I. Table 2 lists example widget schemas and constraints.

EXAMPLE 6. *Consider the DIFFTREE in Figure 8. A radio list has schema <_>, where _ matches any schema or type expression, and is compatible with each* `ANY` *node's schema. The schema mapping for each* `ANY` *node would be* (ANY.v ⟶ radio.v).

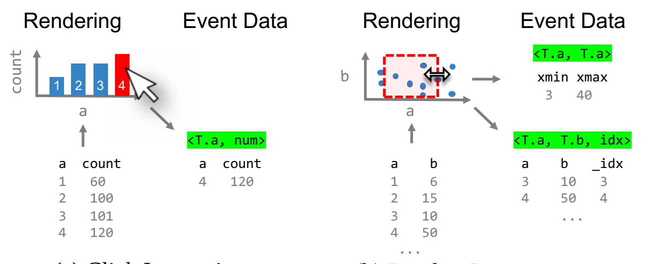
Alternatively, A range slider has schema `<s:num, e:num>`*. The* `list` *node in Figure 8(b) has schema* `<a1:T.a,a2:T.a>`*, which is compatible with the range slider's schema because* T.a *is numeric. The schema mapping is* (a1 ⟶ s, a2 ⟶ e)*. Note that since* `list` *is not a choice node, the event tuples generated by the range slider that are bound to the node will be routed to its child* `ANY` *nodes. Thus, the two* `ANY` *nodes are bound by this interaction mapping.*

The second criteria for a valid mapping is that δ's query bindings satisfy the interaction's constraints. For instance, the range slider has the constraint that the start value s is ≤ to the end value e. We can see that the query bindings for the `list` node in Figure 8 are (1, 3), (2, 4), and satisfy the constraints. Thus, it is valid to map the `list` node to the range slider.

Finally, a mapping is *safe* if there exist user manipulations to produce event tuples for each of the dynamic nodes' query bindings. For widgets, safety is ensured by construction because each widget is initialized with the dynamic node's query bindings. Safety is not always guaranteed for visualization interactions, and we discuss this below.

Widgets: PI_2 is prepopulated with a library of common widgets, including button, radio list, checkbox list, dropdown, slider, range slider, adder, and textbox. For space reasons, Table 2 lists a subset of their schemas and constraints.

Visualization Interactions: A visualization is modeled as a one-to-one projection of input records to marks rendered on the screen.



a	count
1	60
2	100
3	101
4	120

a	count
4	120

a	b
1	6
2	15
3	10
4	50
...	

xmin	xmax
3	40

a	b	_idx
3	10	3
4	50	4
	...	

(a) Click Interaction (b) Brush-x Interaction

Figure 9: Examples of visualization interactions and their event data. A visualization maps each input record to a mark in the chart. User manipulations generate one or more event streams. Event schemas are `in green`.

There are three concerns when modeling interactions in visualizations: (1) each visualization type can support multiple interaction types (e.g., click, brush, pan), (2) each interaction can generate multiple event streams during user manipulations, and (3) the schemas of the event data depend on the visualization's own mapping (Section 4.1). We illustrate these concerns using the bar chart and scatterplot in Figure 9:

Example 7. *The bar chart renders its input four records using the visualization mapping* (a $\longrightarrow$ x, count $\longrightarrow$ y). *One of the interactions that a bar chart supports is click interactions. For instance, if the user clicks the fourth bar, it corresponds to selecting the fourth input record, and thus the event stream emits* (4, 120) *with the schema* `<T.a, num>`.

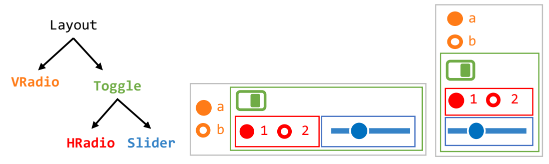
The scatterplot renders its input records using the mapping (a $\longrightarrow$ x, b $\longrightarrow$ y). *It illustrates a 1-D brush interaction (along the x-axis) that emits two event streams. The first stream represents the minimum and maximum bounds of the selection box and has schema* `<T.a,T.a>`, *while the second stream represents the set of selected records, and thus has the same schema as the input data. PI$_2$ internally tracks the index of each record, which is useful for binding* `ANY` *nodes.*

Interaction schemas can depend on the visualization mapping. If the scatterplot used the mapping (b $\longrightarrow$ y, a $\longrightarrow$ x), *then the first stream's schema would instead be* `<T.b, T.b>`.

To this end, each visualization type defines a set of interactions and the event stream schemas for each interaction. The schemas are specified in terms of the visualization's visual attributes, and PI$_2$ uses the visualization mapping to automatically translate them to be in terms of DiffTree's result schema. Thus, each visualization mapping corresponds to a set of interaction event schemas that are candidates for interaction mapping. Table 1 summarizes the interactions each visualization type supports.

4.2.2 Visualization Interaction Safety. Visualization interactions introduce a unique safety concern because their input data is based on the result of a DiffTree. In contrast to widgets, whose domains are initialized based on query bindings in the DiffTree and thus guaranteed to be safe, the values that a visualization interaction can express depends on the contents of its input data. Concretely, consider the bar chart's click interaction in Figure 9. The output of its DiffTree Δ contains four records, thus the click interaction can express the T.a values: 1, 2, 3, 4. Suppose there is a choice node `VAL(4,5)` with schema `<num>`. Based on the above rules, it is valid to map this choice node to the click interaction because their schemas match. However, this specific chart *cannot express the query binding* 5!

We use a simple heuristic to check safety. Given a candidate interaction mapping to an interaction in visualization V, we can check V's visualization mapping Δ $\longrightarrow$ V. Since we know the subset of input queries that Δ expresses, we can (logically) instantiate the visualization with each query's result table, and check the subset of query bindings that the interaction can express. If there exists an input query that can express every query binding, then the interaction mapping is safe. This heuristic appears effective in practice, however as we see in the runtime experiments, checking safety for every candidate mapping degrades PI$_2$ runtime when there are many input queries. We leave optimizations to future work.



(a) Layout Tree (b) Horizontal (c) Vertical

Figure 10: Layout tree and their bounding boxes when the layout nodes (`Layout`, `Toggle`) are horizontal or vertical.

4.3 Layout Mapping $\mathbb{L}$

After visualization and interaction mapping, we can mark the dynamic nodes in the DiffTrees that correspond to widgets on the screen. $\mathbb{L}$ defines a layout tree over the visualizations and widgets, where each layout node either horizontally (`H`) or vertically (`V`) lays out its child elements.

Let a DiffTree node that has been mapped to a widget be called a *Widget Node*. For a given DiffTree Δ, we first create a layout tree W_Δ for its widgets: each widget is a leaf node, and we create a layout node for the least common ancestor of every pair of widget nodes in the DiffTree. Δ's layout tree L_Δ is a layout node whose children are W_Δ and the DiffTree's visualization. The final layout tree $\mathbb{L}$ is a root layout node whose children is each DiffTree's layout tree. Note that layout is currently best effort, users are free to change the widget positioning themselves.

One point of note is that some widgets may themselves be layout nodes. For instance, a radio list, toggle, or tab list may be used to choose different sub-interfaces, but also require space to render the widget themselves. Typically, these "layout widgets" will be mapped to choice nodes that have descendant choice nodes, and thus takes the place of a layout node.

Finally, PI$_2$ also uses the layout tree to estimate the bounding box of every node in the tree. This is used during interface cost estimation to assess the amount of effort that the user must move between widgets and visualizations in order to express a given query, and penalize interfaces that exceed an optional screen size. To do so, we also estimate text and widget sizes based on their initialization parameters.

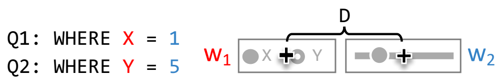


Figure 11: User navigates from w_1 to w_2 to change Q1 to Q2 .

Example 8. *Figure 10 illustrates a layout tree with three leaf widgets, the toggle layout widget, and a root layout node. Vradio (Hradio) renders its elements vertically (horizontally), and the toggle button is always in the top left above its children. Figure 10(a) and (b) illustrate the layout and bounding boxes if the layout nodes were horizontal or vertical, respectively.*

5 COST MODEL

The space of possible interfaces is very large, and a cost model is needed to estimate the quality of a candidate interface mapping. This is deep and long standing problem in HCI research, and a common measure is based on the expected time the user will take to perform the manipulations needed to complete a set of tasks using an interface [7, 15, 24, 45]. PI_2 uses a simple cost model $C(\mathbb{I}, Q) = C_U(\mathbb{I}, Q) + C_L(\mathbb{I})$ that combines usability and layout characteristics of an interface. For usability, we estimate C_U as the time needed for the user to express the sequence of input queries using the interface. For layout, C_L accounts for the interface size.

Usability: We measure usability based on SUPPLE [15], which models the interface cost $C_U(\mathbb{I}, Q) = C_m(\mathbb{I}, Q) + C_{nav}(\mathbb{I}, Q)$ as the time to manipulate each widget (or visualization) C_m and the time to navigate between interactions c_{nav}.

Manipulation cost $C_m(w)$ for a widget w is modeled as a second order polynomial $C_m(w) = a_0 + a_1|w.d| + a_2|w.d|^2$, where $|w.d|$ is the size of the widget's domain. Widgets that enumerate options (e.g., radio, dropdown, and checkboxes) define $|w.d|$ as the number of options; other widgets set $|w.d| = 0$. The manipulation cost for the interface $C_m(\mathbb{I})$ is the total cost of manipulating the widgets needed to express each input query. Our prototype uses parameters fit to widget interaction traces in prior work [9, 57], and sets visualization interaction costs to low constants to encourage choosing them.

The navigation cost C_{nav}, proposed in SUPPLE [15], is based on Fitts' law [27, 45], a model of human movement widely used in HCI and ergonomics. It states that the time to move to a target area increases with its distance D and inversely to its width W along the axis of motion, and has been shown to apply to digital cursors as well as human movement: $a + b \cdot \log_2 2D/W$.

Given the bounding boxes of two widgets, we estimate D as the distance between their centroids, and W as the minimum of the target widget's box width and height [27].

Example 9. *Figure 11 shows two truncated queries that differ in the left and right operands of the equality predicate. To express* `Q1`*, the user manipulates w_1 and w_2 to select* `X` *and* `Y`*, and again to express* `Q2`*, and navigates from $w_1 \to w_2 \to w_1 \to w_2$. In the Fitts' law model, D is the distance between the + markers, and W is set of the box heights. Our prototype sets $a = 1$ and $b = 25$ based on manual experimentation.*

We compute the navigation cost needed to express the input queries in sequence, and navigate the widgets in order of their depth first traversal in the Difftrees.

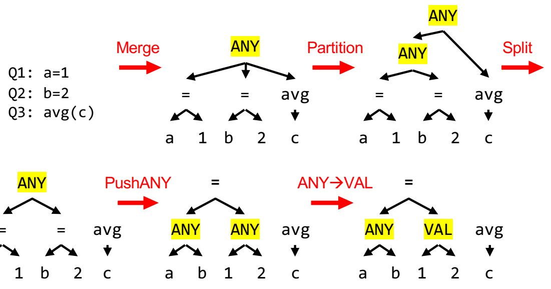


Figure 12: Example sequence of transformation rules applied to three input queries.

Layout: The navigation cost implicitly takes the layout into account, as excessively wide or tall interfaces will require more costly navigation. Thus, by default we set the layout cost $C_L = 0$. However, if the user specifies a maximum desired width and height, then we add a penalization term $C_L(\mathbb{I}) = \alpha * (\max(0, \mathbb{I}.w - width) + \max(0, \mathbb{I}.h - height))$ if the interface size exceeds the desired maximum.

6 INTERFACE GENERATION

As we saw in Section 2, the same set of queries can be expressed by many Difftree structures, each of which can be mapped to many possible interfaces. Given the definitions in the previous sections, we can now present the interface generation problem:

Problem 1 (Interface Generation Problem). *Given input query sequence Q and interface cost model $C(\mathbb{I})$, return an interface mapping $\mathbb{I}^* = (\mathbb{V}, \mathbb{M}, \mathbb{L})$ such that $\mathbb{I}$ expresses Q and minimizes $C(\mathbb{I})$.*

We solve this problem using a search-based approach, where we initialize a set of Difftrees, and iteratively transform the Difftrees and map them to candidate interfaces. This section describes the set of transformation rules that defines the search space, and then describes the search procedure based on Monte Carlo Tree Search (MCTS) [10].

6.1 Difftree Transformation Rules

We defined four categories of Difftree transformation rules that define the search space. Each rule takes as input a choice node and transforms the subtree rooted at the node. All rules are guaranteed to preserve or increase the expressiveness of the Difftrees; since the initial set of Difftrees directly corresponds to the input queries, any reachable set of Difftrees can also express those queries.

Figure 13 shows all the rules. In the diagram, `x`, `y`, `z` represent subtrees that are distinguished by their root node — the root of `x` and `x'` are the same, and different than the `y`'s. `A` represents a AST node and *choice* represents *Choice nodes.*

Each rule category serves a different purpose. *Refactoring rules* identify and refactor shared substructures in order to isolate the precise differences between the queries; these include `PushANY`, `PushOPT1`, `PushOPT2`, and `Partition`. `PushANY` pushes `ANY` nodes down if their children have the same root node and introduces new `ANY` or `OPT` to express the differences between the children's children. `Partition` which groups subsets of an `ANY` node's children, and `PushOPT1` and `PushOPT2`. `PushOPT1` pushes `OPT` node down

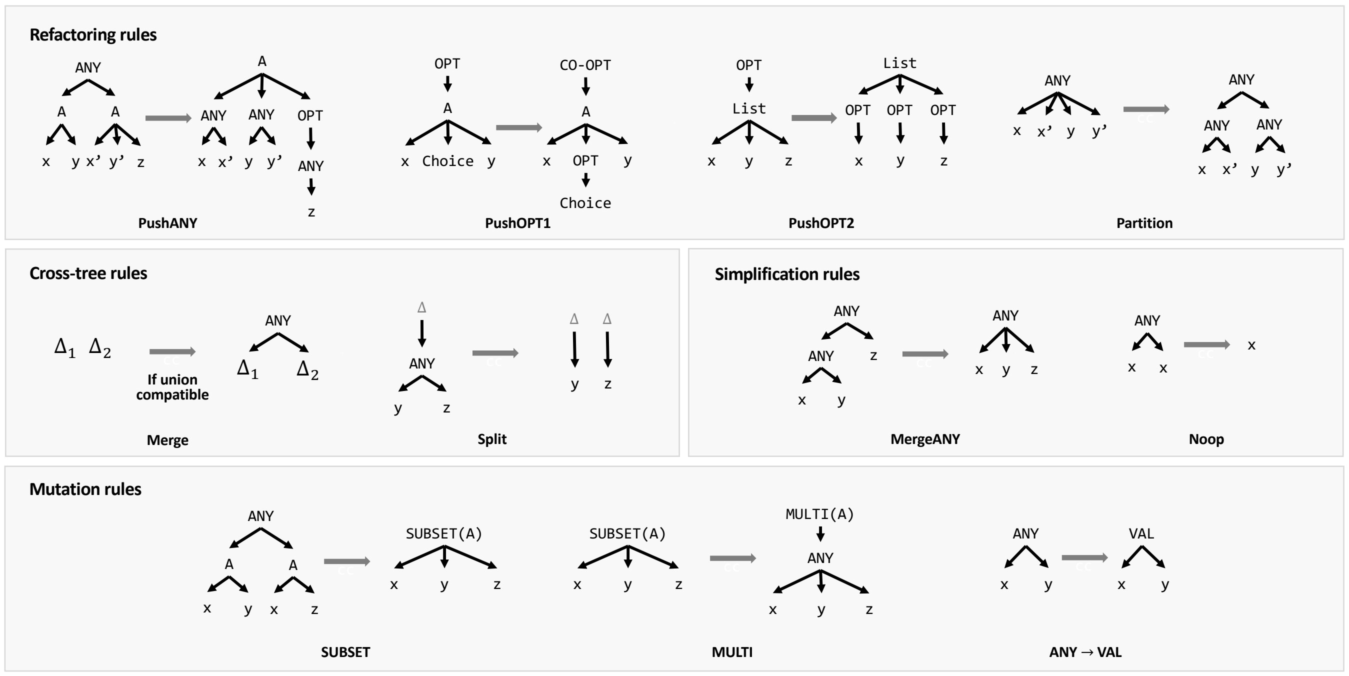


Figure 13: Four categories of Difftree transformation rules that define the search space.

to the *Choice* node and leaves a new `CO-OPT` at the original place indicating that only if the `OPT` exists, the subtree rooted at `CO-OPT` exists. `PushOPT2` pushes the `OPT` down to all the children of the list node, which increases the expressiveness of the Difftrees. *Cross-tree rules* are used to `Merge` multiple Difftrees into one if their or `Split` one Difftree into multiple. *Mutation rules* transform one type of choice node into another, for instance `ANY` $\longrightarrow$ `VAL`, `MULTI`, and `SUBSET` in Figure 13 . Finally, *Simplification rules* are used to simplify the tree structure, for instance `Noop` removes `ANY` nodes with a single unique child, and `MergeANY` reduces a cascade of `ANY` nodes into a single one.

Example 10. *Figure 12 applies a sequence of transform rules on three input query fragments. Each query is initially a separate Difftree, and* `Merge` *combines them into a single Difftree.* `Partition` *groups the* `ANY`*'s children into homogenous clusters, and combines each non-singular cluster with an* `ANY`*. In practice,* `Partition` *is used to initially cluster the input queries by their result schema in order to reduce the number of redundant visualizations and maximize the likelihood of non-tabular visualization mappings.* `Split` *then removes the root* `ANY` *so that its children are separate Difftrees. Since both equality predicates are rooted with* =*,* `PushANY` *pushes the* `ANY` *down. Finally, the* `ANY` *on the right has numeric children, so it is lifted to a* `VAL`.

6.2 Monte Carlo Tree Search

A search problem is defined by its states, transitions, and cost function. In our problem, each set of Difftrees is a state, and transform rules define the transitions. To assess the quality of each state, we apply the cost model in Section 5 to a set of K random interface mappings. The search returns a set of Difftrees, and we perform a more complete search for the final interface mapping $\mathbb{I}$.

Since the number of applicable transform rules is very large, it is infeasible to exhaustively search even a small portion of the search space. Thus, we adopt Monte Carlo Tree Search [10] (MCTS), a randomized search algorithm famously applied to problems with massive search spaces, such as Google's AlphaGo [46]. MCTS balances exploration of new states with exploitation of known good states. Another benefit is that MCTS works well when high-cost states are needed in order to reach an optimal state, as is common in games like Go. Since MCTS is traditionally used in two-player games, we use a single-player variation [42].

6.2.1 MCTS Search Procedure. Single-player MCTS [42] explores the search space by iteratively growing a *search tree*, where each node is a search state[2]. Each iteration grows the tree using the following four steps. Note that we add a special `TERMINATE` rule that is a valid transition for every state. Choosing this rule results in a terminal state that has no outgoing transitions. The algorithm stops when all leaves reach a terminal state.

(1) **Select** a leaf state in the search tree. Starting from the root state R, we recursively select a child until we reach a leaf state S. Child selection is based on the Upper Confidence Bound for Trees strategy (UCT) [42]: let N_i be the ith child of current state N, we choose the child that maximizes

$$\overline{X} + c\sqrt{\frac{\ln t(N))}{t(N_i)}} + \sqrt{\frac{\sum x^2 - t(N_i)\overline{X}^2 + d}{t(N_i)}} \quad (1)$$

[2] *State* refers to a search tree node in order to distinguish from a Difftree node.

Where t(N) is the number of times that a node N was visited. This estimates an upper confidence bound for the expected reward $\bar{X}$, which is the negative cost. The term exploits the existing reward, the second term prefers the unexplored nodes, and the third term prefers nodes with high reward variance. c and d are empirically set constants and control the preference for exploration and high variance nodes, respectively.

(2) **Expand** the leaf S by adding all of its children (result of valid transform rules) to the search tree (their visit count would be 0, which will prioritize them in future iterations). Then randomly choose a child state C to simulate.

(3) **Simulate** a random playout by applying random transform rules to C until there are no more valid transform rules to apply, or if the `TERMINATE` rule is chosen.

(4) **Backpropagate** the leaf node's reward from C to R by incrementing the visit count and appending the reward to all states in its path. We estimate the reward by generating $K = 5$ random interface mappings, estimating their costs, and returning the negative of the minimum cost (and thus the maximum reward).

In addition, we developed a set of optimizations that work well in practice. The first is when choosing the best Difftree to return once the search terminates. MCTS traditionally returns the state with the highest average reward. In contrast, we follow Cadiaplayer [4] and return the state with the maximum reward encountered (during its rollouts and random interface mappings).

The second is to run the search iterations in parallel. Every s iterations, the coordinator synchronously receives the highest reward state from each worker and distributes the maximum reward state back to the workers. We further use early stopping, where each worker sends an early stop signal if its local optimal state has not changed in es iterations. If the coordinator receives stop signals from all workers and does not receive a higher reward state, it terminates the search. Section 7.3 evaluates these optimizations.

6.2.2 Interface Mapping Generation. Given the output state from MCTS, we perform a more exhaustive search for the lowest cost interface mapping. This is done in three phases, where first, we enumerate all possible visualization mappings $\mathbb{V}$, and then derive all the valid visualization interactions over these visualizations. Afterwards, we search for all the mappings from the *choice nodes* to the widgets and visualization interactions $\mathbb{M}$. Finally, we construct the layout tree and use a prior branch-and-bound-based algorithm [15] to assign horizontal and vertical layouts to each layout node. Since manipulation cost C_m term in the cost model is independent of the layout, and typically the dominant factor, we separate the search into two steps: first, we search for the top k mappings of $\mathbb{V}, \mathbb{M}$ in terms of C_m, and second, we search the optimal layout for each of these top k mappings. In the end, we return the overall optimal interface. We find $k = 10$ is sufficient to find the optimal interface empirically.

Algorithm 1 shows the pseudo code of our search algorithm for $\mathbb{V}, \mathbb{M}$. At first, we enumerate $\mathbb{V}$ as shown in the `searchV` function(line 19). Then, we consider $\mathbb{M}$. An interaction mapping $\mathbb{M}$ is valid if and only if all the interactions exactly express the *choice nodes* once – namely, to find the *exact cover* of the *choice nodes* using widgets or visualization interactions. Notice that, the attribute

Algorithm 1: $\mathbb{V}$, $\mathbb{M}$ Mapping Generation Algorithm

```
   Result: Given Difftrees Δ and queries Q, find the top k
           mappings of $\mathbb{V}$ and $\mathbb{M}$ with lowest $C_m$.
1  clist := an ordered list of choice nodes in Δ;
2  icand := each choice node's all valid visualization
     interaction candidates;
3  wcand := each choice node's all valid widget candidates ;
4  G[N] := the lowest $C_m$ among all widget covers of N;
5  F[N] := the top k exact widget covers of N with lowest $C_m$;
6  minHeap := the top k mappings of $\mathbb{V}$ and $\mathbb{M}$ with lowest $C_m$;
7  Function G(N):
8      if G[N] hasn't been calculated then
9          G[N] = min($C_m$(w, Q) + G (N-w.cover) | w ∈
             wcand[[N[0]]])
10     return G[N]
11 Function F(N):
12     if F[N] hasn't been calculated then
13         L = ∅;
14         for w in wcand[N[0]] do
15             if w.cover ⊆ N then
16                 L = L ∪{C ∪ {w.cover → w}
                     |C ∈ F(N − w.cover)}
17         F[N] = top K elements in L with lowest costs
18     return F[N]
19 Function searchV(Δ, $\mathbb{V}$, $\mathbb{M}$):
20     for v in all possible visualization mapping: do
21         $\mathbb{V}$ = v;
22         compute icand;
23         searchM(Δ, 0, $\mathbb{V}$, $\mathbb{M}$ );
24 Function searchM(Δ, i, $\mathbb{V}$, $\mathbb{M}$):
25     N := the choice nodes without mapping in clist[0:i];
26     Ns := all the choice nodes without mapping;
       /* pruning */
27     if $C_m$({$\mathbb{V}$, $\mathbb{M}$}, Q) + G (N) >= minHeap[k].cost then
28         return
29     if i == len(clist) then
30         for m in F (N) do
31             $\mathbb{M}$.add (m);
32             if $C_m$({$\mathbb{V}$, $\mathbb{M}$}, Q) < minHeap[k].cost then
33                 insert into minHeap;
34             $\mathbb{M}$.delete (m);
35         return
36     for vinteraction in icand[i] do
37         if vinteraction.cover ⊆ Ns and compatible with $\mathbb{M}$
             then
38             $\mathbb{M}$.add (vinteraction);
39             searchM (Δ, i + 1, $\mathbb{V}$, $\mathbb{M}$);
40             $\mathbb{M}$.delete (vinteraction);
41     searchM (Δ, i + 1, $\mathbb{V}$, $\mathbb{M}$) ;
42 searchV (Δ, ∅, ∅);
43 return minHeap
```

cover(line 15) of a choice node's candidate interaction means besides this choice node, all the choice nodes it expresses at the same time. For example, in Figure 8, both ANY have RangeSlider as its candidate widget and RangeSlider.cover are these two ANY. Also, visualization interaction mapping is more sophisticated in that ① one visualization interaction can not be mapped to multiple times in the same DIFFTREE because it will decrease the expressiveness; ② on one visualization, some interactions are conflicted, such as brush along x-axis and brush along y-axis, so that only one of them can be chosen. With such concerns, the interaction mapping can not be solved by single dynamic programming. Thus, we separate visualization interaction mapping and widget mapping – first, enumerate the compatible visualization interactions (line 36); and for each visualization interaction mapping, find the optimal widget mappings(line 30) for the left uncovered choice nodes using dynamic programming in F(N)(line 11). Also, we prune the search space by proposing a lowest bound(line 27), which equals to the existing visualization interaction mapping's cost plus the lowest possible widget mapping cost for uncovered choice nodes computed by G(N)(line 7). When this lowest bound is greater than the existing k_{th} optimal cost, we prune this branch.

7 EXPERIMENTS

The primary success criteria for PI$_2$ is to generate fully functional interactive visualizations from a small number of input examples. We break this down into three questions. 1) Can PI$_2$ generate expressive interactive visualization interfaces? To evaluate this, we follow the evaluation in Vega-lite [41] and show examples that cover the data-oriented interactions in Yi et al.'s [55] taxonomy of interaction methods. We further show that PI$_2$ can reproduce the COVID visualization shown at the top of Google's search results page for the search query "covid19". 2) How PI$_2$ help interface creation in realistic settings? We illustrate this by using a subset of queries from the Sloan Digital Sky Survey [56] (SDSS) to produce a custom interface. We also show a case study to create an analysis dashboard for the Kaggle supermarket sales dataset [22] from complex sales analysis queries that existing authoring tools (e.g., Metabase, Tableau) do not support. 3) How quickly can PI$_2$ generate interfaces? We evaluate PI$_2$'s latency and generated interface quality varies with respect to its search parameters.

Our focus is on the *functionality* of the output interface (whether it can easily express the underlying analysis) rather than its style and presentation. Thus, when applicable, we may modify the font, spacing, and other CSS-based interface stylings.

7.1 Interaction Expressiveness

We use Yi et al.'s [55] taxonomy of visualization interaction techniques to highlight PI$_2$'s expressiveness. Their paper describes seven interaction types: *Select* interesting data; *Explore:* show different subsets of the data; *Abstract:* change the level of detail; *Filter* the data; *Connect:* highlight related data (such as in a different chart); *Encode:* change the visual representation (e.g., from scatterplot to bar chart); *Reconfigure:* rearrange the visual presentation (e.g., change from linear to log scale) Of these, encode and reconfigure are unrelated to query-level transformations. Every example supports selection, so we evaluate the remaining four types.

In each example, we **highlight** query fragments that differ from the preceding query, use .. when a long substring does not change, and use BTWN min & max to mean BETWEEN min AND max.

```
Q1  SELECT hp, mpg, origin from Cars
     WHERE hp BTWN 50 & 60 AND mpg BTWN 27 & 38
Q2 ..WHERE hp BTWN 60 & 90 and mpg BTWN 16 & 30
```

Listing 1: Explore

Explore: The queries in Listing 1 project horsepower (hp), miles per gallon (mpg), and origin from the Cars dataset, and change the range predicates on hp and mpg. The generated interface in Figure 14a renders the attributes as the x-axis, y-axis, and color, respectively, and enables panning and zooming to control the range predicates (thus it also satisfies *Abstract* interaction described next).

```
Q1 SELECT date, price FROM sp500
Q2 ..WHERE date > '2001-01-01' AND date < '2003-01-01'
Q3 ..WHERE date > '2001-02-01' AND date < '2003-02-01'
```

Listing 2: Abstract

Abstract The queries in Listing 2 also vary a range predicate, however Q1 does not have a WHERE clause. The overview-and-detail interface (Figure 14c) has a static overview chart for Q1; brushing it updates the detail line chart that expresses the filtered data.

```
Q1 SELECT hp, disp, id FROM Cars
Q2 SELECT mpg, disp, id in (1, 2) as color FROM Cars
Q3 SELECT mpg, disp, id in (20,22) as color FROM Cars
```

Listing 3: Connect

Connect: Q1 in Listing 3 returns horsepower (hp), displacement (disp), and id from the Cars dataset, while the latter queries return miles per gallon (mpg), and a boolean color attribute based on the ids of the rows. In Figure 14b, the left scatterplot renders hp and disp as the x and y axes (id is a primary key so is not rendered by default), while the right scatterplot also encodes the boolean color attribute as the mark color. Thus, selecting points in the hp chart highlights the corresponding rows in the mpg chart.

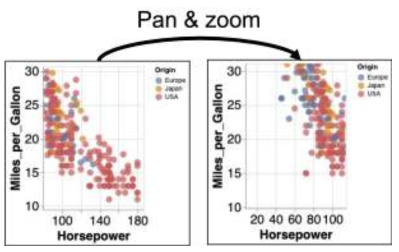
Filter: Listing 4 lists three sets of queries; each is grouped by a different attribute (hour, delay, and dist). Q1,4,7 do not filter the table, and the subsequent queries filter by the grouping attributes of the other two sets. For instance, Q2,3 group on hour and filter on delay and dist. These queries describe cross-filtering, which PI$_2$ automatically derives from first principles (Figure 14d). Brushing in a chart updates the corresponding predicates in *both* other charts, and clearing the brush disables the predicate (toggles its existence).

```
Q1 SELECT hour,count(*) FROM flights GROUP BY hour
Q2 ..WHERE delay BTWN 0 & 50 AND dist BTWN 400 & 800..
Q3 ..WHERE delay BTWN 10 & 60 AND dist BTWN 10 & 300..

Q4 SELECT delay,count(*) FROM flights GROUP BY delay
Q5 ..WHERE hour BTWN 10 & 16 AND dist BTWN 400 & 800..
Q6 ..WHERE hour BTWN 15 & 20 AND dist BTWN 200 & 700..

Q7 SELECT dist, count(*) FROM flights group by dist
Q8 ..WHERE hour BTWN 10 & 16 AND delay BTWN 0 & 50..
Q9 ..WHERE hour BTWN 8 & 19 AND delay BTWN 20 & 61..
```

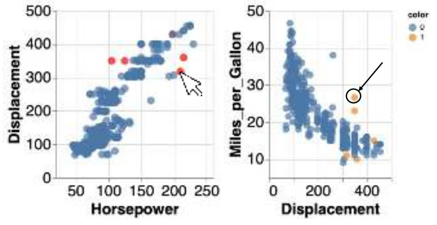
Listing 4: Filter

7.2 Case Studies

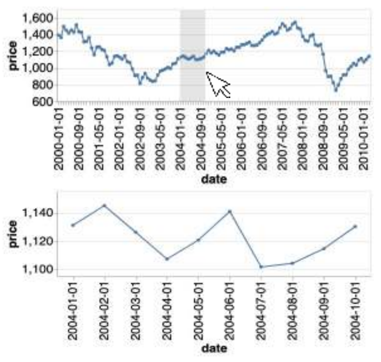
We show case studies that use a real-world query log, reproduce a real-world visualization, and author a complex sales dashboard.



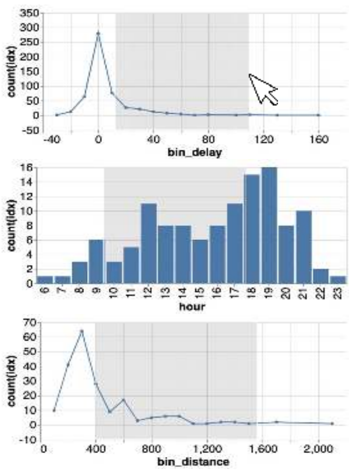
(a) Explore



(b) Connect

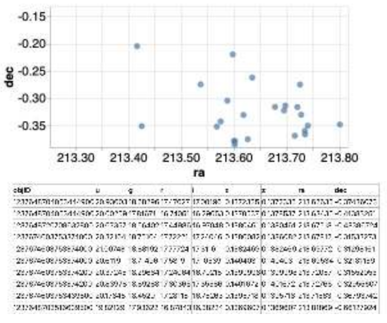


(c) Abstract

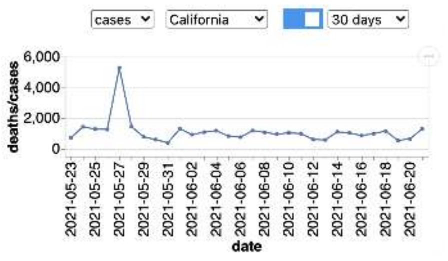


(d) Filter

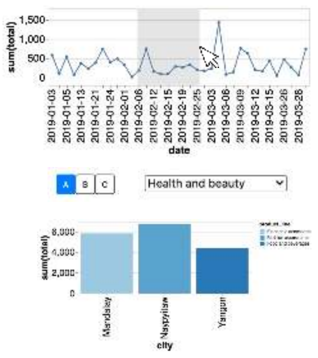
Figure 14: Interfaces that express Yi et al.'s [Yi et al.] interaction taxonomy using queries in Listing 1, Listing 3, Listing 2 and Listing 4. (a) panning ans zooming interaction, (b) linked selection, (c) overview and details interaction, (d) cross-filtering.



(a)



(b)



(c)

Figure 15: Interfaces generated for case studies. (a) new interface from real SDSS queries, (b) reproducing Google's Covid-19 Vis, (c) authoring a custom sales analysis dashboard.

```
Q1 SELECT DISTINCT gal.objID, gal.u, gal.g, gal.r,
          gal.i, gal.z, s.z, s.ra, s.dec
   FROM galaxy as gal, specObj as s
   WHERE s.bestObjID = gal.objID AND s.z BTWN 0.1362 & 0.141 AND
          s.ra BTWN 213.3 & 214.1 AND s.dec BTWN -0.9 & -0.2
Q2 ..AND s.ra BTWN 213.4191 & 213.9 AND s.dec BTWN -0.565 & -0.3111
Q3 ..AND s.ra BTWN 213.5 & 213.8 AND s.dec BTWN -0.34 & -0.2
       -- many similar queries --
Q8 select DISTINCT ra, dec FROM specObj
     WHERE ra BTWN 213.2 & 213.6 AND dec BTWN -0.3 & -0.1
Q9 ..WHERE ra BTWN 213 & 214 AND dec BTWN -0.8 & -0.4
```

Listing 5: Subset of SDSS queries

SDSS queries: Visitors can use textbox-based forms on the SDSS website [43] to select a subset of stars that are returned as a text table. The non-interactive forms are complex to support a wide range of analyses. PI_2 uses a subset of SDSS queries [44] (Listing 5) to generate a custom analysis interface. Q1 is a join query to filter stars by their celestial coordinates (z, ra, dec) and Q2-7 vary the right ascension (ra) and declination (dec). Finally, Q8,9 return star locations within a bounding box. Figure 15a renders the first set of queries as a table because those queries return 9 attributes, and renders the star locations (Q8,9) as a scatterplot. Users can pan and zoom in the scatterplot to update the table. In short, PI_2 transformed a text-based form into a fully interactive visual analysis interface.

```
Q1 SELECT date, cases FROM covid WHERE state='CA'
Q2 ..WHERE state='WA' and date>date(today(), '-30 days')
Q3 ..WHERE state='CA' and date>date(today(), '-7 days')
Q4 SELECT date, deaths FROM covid WHERE state='CA'
Q5 ..WHERE state='NY'
Q6 ..WHERE state='WA' and date>date(today(),'-14 days')
Q7 ..WHERE state='WA' and date>date(today(),'-7 days')
Q8 ..WHERE state='NY' and date>date(today(),'-7␣days')
```

Listing 6: Covid visualization queries

Reproducing Google's Covid-19 Vis This example uses queries in Listing 6 to reproduce the Covid-19 visualization on Google's results page for "covid19". Q1-3 compute daily confirmed cases for different states and date intervals, while Q4-8 report daily deaths. Note that Q1,Q4 do not filter by date. Figure 15b reproduces the interactions in Google's visualization. The dropdowns change the reported metric, state filter, and date interval. The latter is an example of a nested interaction, because the filter on date interval dropdown is only enabled when the toggle is turned on.

```
Q1    SELECT city, product, sum(total) FROM sales as ss
      WHERE ss.date
      GROUP BY city, product
      HAVING sum(total) >= ( SELECT max(t) FROM
        (SELECT sum(total) as t FROM sales as s
         WHERE s.city = ss.city and
         GROUP BY s.city, s.product ) )
Q2 ..WHERE ss.date BTWN '2019-01-25' & '2019-02-15'
..HAVING sum(total) >= ( SELECT max(t)
            ..s.date BTWN '2019-01-25' & '2019-02-15' ..'
Q3␣..WHERE␣ss.date␣BTWN␣'2019-01-25' & '2019-02-15'
   ..HAVING sum(total) >= ( SELECT max(t)
            ..s.date BTWN '2019-01-25' & '2019-02-15' ..
Q4    SELECT date, sum(total) FROM sales
      WHERE branch = 'A' AND product = 'Health and beauty'
      GROUP BY date
Q5    ..WHERE branch = 'B' and product = 'Electronics'..
Q6    ..WHERE branch = 'C' and product = 'Lifestyle'..
        -- many similar queries --
```

Listing 7: Complex sales analysis queries

Authoring a Sales Dashboard: This example is an analysis that current authoring tools can't create. Query 1 in Listing 7 returns the total sales for products in different cities with the maximum total sales; it has multiple nested queries in the HAVING clause. Q2 modifies the query by specifying the date range, Q3 modifies Q2's date predicate in the outer and nested queries, and the remaining return total sales by date for different branches and products. The top chart in Figure 15c renders the total sales by date, with controls to filter by branch and product; brushing it updates the bar chart (which renders the product with top sales for each city during the time period specified by the brush). PI$_2$ can transform arbitrarily complex queries, and link visualizations. Existing authoring tools cannot generate this interface: Metabase only supports parameters in the WHERE clause [32], and Tableau does not parameterize custom queries [49].

7.3 PI$_2$ Performance and Quality

PI$_2$ is primarily affected by three parameters:

Early Stop: stop MCTS if the optimal DIFFTREE doesn't change in es $\in [5, 100]$ iterations (default 30).

Parallelize over p $\in [1, 4]$ workers (default 3).

Synchronization Interval: every s $\in [5, 100]$ MCTS iterations (default 10).

We measure runtimes for MCTS search and the final interface mapping separately. We report the interface quality as follows: given c as an interface's cost, and c^* as the minimum cost over all evaluated conditions for a query log, we report $\frac{c^*}{c}$. 1 means the generated interface is optimal, and worse interfaces converge towards 0. We visually verified that interfaces with c^* were indeed the best, and qualitatively, interfaces with quality above 85% are nearly the same as the optimal (see appendix for examples of interfaces of varying qualities).

We used all 7 query logs above, and average over 10 runs/condition on 4x2.2GHz 16GB RAM Google Cloud VMs with Ubuntu 20.04 LTS.

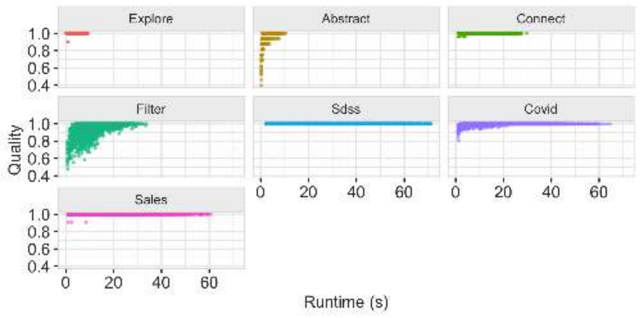


Figure 16: Runtime-quality trade-offs across all conditions.

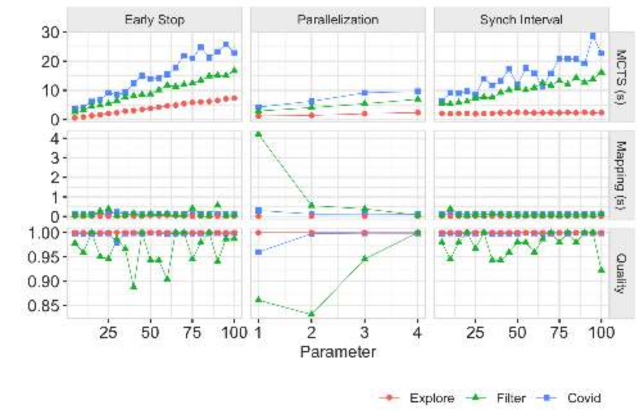


Figure 17: Varying early stop and synchronization interval increases runtime without tangible improvements to interface quality. In contrast, parallelization increases the MCTS cost, but can find higher quality DIFFTREE structures for complex interfaces such as Filter.

Runtime-Quality Trade-off: We first report the end-to-end runtime and interface quality from a sweep of all the parameters. We vary es and s between 5 to 100 in increments of 5, and vary the parallelization from 1 to 4. We find that PI$_2$ is able to find the optimal interface in less than 1 second for the "simpler" query logs (e.g., Explore, Connect, Sales, SDSS). Filter and Covid are more challenging because they result in many visualizations or interactions, and we see a general trade-off where configurations with longer runtimes tend to generate higher quality results. We will dive into the sensitivity to each parameter below. To avoid crowding the graphs, we will use Explore to represent the "simpler" logs.

Parameter Sensitivity Figure 17 reports the three metrics (rows) while varying each parameter (cols). To keep the plots legible, we report results for Explore, Filter, and Covid because the remaining logs have results nearly identical to Explore. Varying early stop and the synchronization interval increases the MCTS runtime, but does not impact interface quality. This is because PI$_2$ finds the optimal DIFFTREE structure very quickly, so larger early stop values or synchronization intervals simply delay MCTS termination.

Varying parallelization (middle col) slows MCTS search due to synchronization overhead and stragglers. The mapping cost for Filter when there is no parallelization is higher because the poor quality Difftree contains a huge number of choice nodes that must be mapped. High quality Difftrees tend to have fewer choice nodes, which reduces the number of redundant interactions. Increasing parallelization improves quality for Filter because MCTS explores a larger subset of the search space. Note that the y-axis for quality ranges from 85% to 100%, thus even with low parallelization, the interfaces have reasonable quality.

Scalability: We also evaluated the runtime as the number of input queries increases from 9 to 900 (by duplicating the Filter log). We find that the runtime increases roughly linearly from a few seconds to ≈2000s for 900 queries. This is expected as PI_2 is not optimized for scalability, and is dominated by (1) increased number of search states, (2) higher cost to estimate the navigation cost due to the larger number of queries, and (3) increased cost to check safety. We expect sampling, caching, and approximation optimizations can reduce these bottlenecks considerably. Further, we note that in an authoring setting, the number of queries is unlikely to be large.

8 RELATED WORK

Interface generation: Existing works either take analysis queries into account, nor generate fully interactive analysis interfaces. For instance, prior work in the DB community generates form-based search and record creation interfaces based solely on the database content [18–20], but may generate over-complex forms because it does not leverage analysis queries. Similarly, techniques from the HCI community rely on the developer to provide task and data specifications [14, 36, 38, 48, 51]. In this sense, PI_2 models the desired task using example queries. Visualization recommendation algorithms [28, 30, 33, 52, 53, 58] output visualization designs based on an input dataset but not full analysis interfaces. PI_1 [57] uses input queries, but is limited to unordered sets of widgets.

Authoring Tools: There are numerous interface [8, 21, 39] and visualization [5, 41] programming libraries available, and tools like AirTable [2], Figma [11], and Plasmic [37] help interface designers rapidly iterate on the interface design. However, these still require programming effort to use and combine. Dashboard authoring tools such as Metabase [31], Retool [40], and Tableau [35] let non-technical users create interactive SQL-based analysis dashboards. However, they restrict types of analyses and queries (e.g., to parameterized literals or OLAP queries) in order to keep their own interfaces simple. In contrast, PI_2 supports arbitrarily complex queries and structure transforms for nearly any syntactic element in the queries. Users "program" PI_2 by providing example queries.

9 DISCUSSION AND CONCLUSION

PI_2 is the first system to generate fully functional interactive visualization interfaces from a few examples analysis queries. PI_2 introduces the Difftree structure to succinctly encode syntactic variations between input queries as *choice nodes*. It formulates interface generation as a schema matching problem from Difftree results to visualizations, choice nodes to interactions, and Difftree structure to layout. Transform rules "refactor" the Difftrees to produce different interfaces. PI_2 can generate interfaces to cover the interaction taxonomy proposed by Yi et al. [55], can generate useful interfaces from real-world query logs, and replicate existing visualizations, and can be used to author visualizations that are not possible in existing visual authoring tools. In the evaluation, PI_2 generated interfaces in 2 – 19s, with a median of 6s.

Our future work plans to improve system usability so users can control how much PI_2 generalizes from the input queries, and replace subsets of the interface they do not like. Further, we will focus on generating informative labels, support design principles such as alignment and spacing, and improve scalability.

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[2] Airtable. 2021. Airtable. https://airtable.com/.
[3] Jacques Bertin. 1983. *Semiology of graphics; diagrams networks maps.* Technical Report.
[4] Yngvi Bjornsson and Hilmar Finnsson. 2009. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games* 1, 1 (2009), 4–15.
[5] M. Bostock, Vadim Ogievetsky, and J. Heer. 2011. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17 (2011), 2301–2309.
[6] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), 1–43.
[7] S. Card, T. Moran, and A. Newell. 1983. The psychology of human-computer interaction.
[8] Winston Chang, Joe Cheng, J Allaire, Yihui Xie, Jonathan McPherson, et al. 2015. shiny: Web Application Framework for R, 2015. In *CRAN.*
[9] Y. Chen and Eugene Wu. 2020. Monte Carlo Tree Search for Generating Interactive Data Analysis Interfaces. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).
[10] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games.*
[11] Figma. 2021. Figma: the collaborative interface design tool. https://www.figma.com.
[12] FiveThirtyEight. 2021. All Posts Tagged Data Visualization. https://fivethirtyeight.com/tag/data-visualization/.
[13] Flask. 2021. Welcome to Flask — Flask Documentation (2.0.x). https://flask.palletsprojects.com/en/2.0.x/.
[14] Krzysztof Z Gajos and Daniel S. Weld. 2004. SUPPLE: automatically generating user interfaces. In *IUI '04.*
[15] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically generating personalized user interfaces with Supple. *Artif. Intell.* 174 (2010), 910–950.
[16] Google. 2021. Facets - Know Your Data. https://pair-code.github.io/facets/.
[17] iCheck. 2021. iCheck. icheckuclaim.org.
[18] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. 2007. Making database systems usable. In *SIGMOD.*
[19] Magesh Jayapandian and HV Jagadish. 2008. Automated creation of a forms-based database query interface. In *PVLDB.* VLDB Endowment.
[20] Magesh Jayapandian and H. V. Jagadish. 2006. Automating the Design and Construction of Query Forms. In *TKDE.*
[21] JQuery. 2021. jQuery. https://jquery.com/.
[22] Kaggle. 2021. Dataset: Supermarket Sales. https://www.kaggle.com/aungpyaeap/supermarket-sales.
[23] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011).
[24] D. Kieras. 1994. GOMS modeling of user interfaces using NGOMSL. In *CHI '94.*
[25] Knex.js. 2021. A SQL Query Builder for Javascript. http://knexjs.org/.
[26] Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. Deepeye: Towards automatic data visualization. In *2018 IEEE 34th international conference on data engineering (ICDE).* IEEE, 101–112.
[27] I Scott MacKenzie and William Buxton. 1992. Extending Fitts' law to two-dimensional tasks. In *Proceedings of the SIGCHI conference on Human factors in computing systems.* 219–226.
[28] Jock Mackinlay. 1986. Automating the design of graphical presentations of relational information. In *Transactions On Graphics.*
[29] J. Mackinlay, P. Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *IEEE Transactions on Visualization and Computer*

Graphics 13 (2007).
[30] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show me: Automatic presentation for visual analysis. In *TVCG*.
[31] Metabase. 2021. Metabase. https://www.metabase.com.
[32] Metabase. 2021. Metabase Documentation: SQL Parameters. https://www.metabase.com/docs/latest/users-guide/13-sql-parameters.html.
[33] Dominik Moritz, Chenglong Wang, Greg L. Nelson, H. Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2018. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco.. In *TVCG*.
[34] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and J. Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics* 25 (2019), 438–448.
[35] Daniel Murray. 2013. Tableau Your Data!: Fast and Easy Visual Analysis with Tableau Software.
[36] Jeffrey Nichols, Brad A Myers, and Kevin Litwack. 2004. Improving automatic interface generation with smart templates. In *IUI*.
[37] Plasmic. 2021. Plasmic: The visual builder for your tech stack. https://www.plasmic.app/.
[38] Angel R Puerta, Henrik Eriksson, John H Gennari, and Mark A Musen. 1994. Model-based automated generation of user interfaces. In *AAAI*.
[39] React. 2021. A JavaScript library for building user interfaces. https://reactjs.org/.
[40] Retool. 2021. Build internal tools, remarkably fast. https://www.retool.com.
[41] A. Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and J. Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23 (2017), 341–350.
[42] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ-B Chaslot, and Jos WHM Uiterwijk. 2008. Single-player monte-carlo tree search. In *International Conference on Computers and Games*. Springer, 1–12.
[43] SDSS. 2021. SDSS Skyserver Website. http://skyserver.sdss.org/dr16/en/tools/search/radial.aspx.
[44] SDSS. 2021. SDSS Weblog SQL Search. http://skyserver.sdss.org/log/en/traffic/sql.asp?url=http://skyserver.sdss.org/log/en/traffic///help/download/.
[45] A. Sears. 1993. Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. *IEEE Trans. Software Eng.* 19 (1993), 707–719.
[46] D. Silver, Aja Huang, Chris J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, S. Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016), 484–489.
[47] C. Stolte, Diane Tang, and P. Hanrahan. 2008. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM* 51 (2008), 75–84.
[48] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.
[49] Tableau. 2021. Connect to a Custom SQL Query. https://help.tableau.com/current/pro/desktop/en-us/customsql.htm.
[50] Tableau. 2021. Cube Data Sources. https://help.tableau.com/current/pro/desktop/en-us/cubes.htm.
[51] Jean Vanderdonckt. 1994. Automatic generation of a user interface for highly interactive business-oriented applications. In *CHI*.
[52] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
[53] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. In *TVCG*.
[54] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining away outliers in aggregate queries. (2013).
[55] J. S. Yi, Y. Kang, J. Stasko, and J. Jacko. 2007. Toward a Deeper Understanding of the Role of Interaction in Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 13 (2007), 1224–1231.
[56] Donald G York, J Adelman, John E Anderson Jr, Scott F Anderson, James Annis, Neta A Bahcall, JA Bakken, Robert Barkhouser, Steven Bastian, Eileen Berman, et al. 2000. The sloan digital sky survey: Technical summary. *The Astronomical Journal* 120, 3 (2000), 1579.
[57] Qianrui Zhang, Haoci Zhang, Thibault Sellam, and Eugene Wu. 2019. Mining precision interfaces from query logs. In *Proceedings of the 2019 International Conference on Management of Data*. 988–1005.
[58] Jonathan Zong, D. Barnwal, Rupayan Neogy, and A. Satyanarayan. 2021. Lyra 2: Designing Interactive Visualizations by Demonstration. *IEEE Transactions on Visualization and Computer Graphics* 27 (2021), 304–314.

A EXAMPLES OF INTERFACES OF VARYING QUALITIES

Below shows two non-optimal interfaces. Figure 18 shows an interface for filter queries in Listing 4 with its quality equal to 0.87. Compared with the optimal interface in Figure 14d, it has an extra toggle button which toggles the existence of the first chart's filter predicates. If the toggle is on, the brushing in the second and third charts will update the first chart by adding corresponding predicates to its underlying query's where clause, otherwise, the first chart will not be affected by the brushing interactions on the other charts. Figure 19 shows the interface for sales analysis queries in Listing 7 whose quality is 0.893. Compared to the optimal one in Figure 15c, the third static chart is newly added which only express `Q1` – the total sales for products in different cities with the maximum total sales *without* any date range constraint. The other charts stay the same that brushing in the first chart will update the second chart by specifying the date range. From above, we can see that although we did not find the optimal DIFFTREES in these two example, such non-optimal interfaces with high quality above 85% are already nearly the same as the optimal ones .

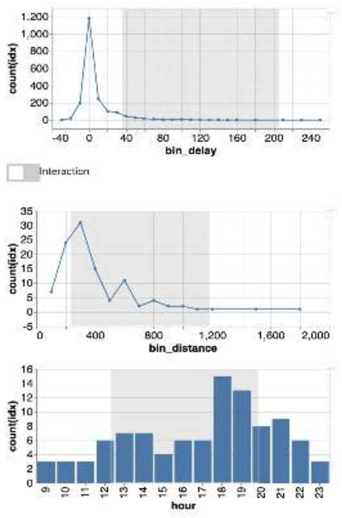


Figure 18: Non-optimal interface for filter usecase with quality = 0.87.

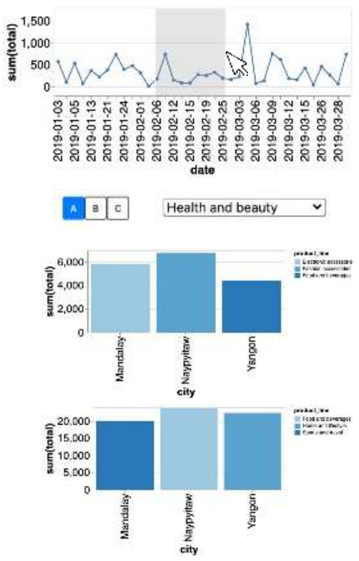


Figure 19: Non-optimal interface for sales analysis queries with quality = 0.893.