DSC-204A: Scalable Data Systems, Fall 2025

Guest Lecture: Systems for Human Data Interaction

Lecturer: Eugene Wu *Scribe: Chen-Fan Lee, Dian Tu, Rachel Wang, Yizhao Chen*

1 Introduction: Lab Background and the Challenge of Designing Data Interfaces

Guest Lecturer's Background: Eugene Wu is a Tenured Associate Professor of Computer Science at Columbia University and Co-Director of the Data Agents and Processes Lab. His research focuses on data management systems, human–data interaction, and data visualization.

Data Agents Processes Lab's Background: The lab's research spans data management, data visualization, human–computer interaction (HCI), and machine learning systems. One of their goals is to develop systems that enhance human–data interaction.

Challenges: While modern tools have made general application development easier, building data interfaces remains difficult. Mr. Wu identified two fundamental challenges:

- **Scale:** As datasets grow from megabytes to terabytes, developers must shift architectures (from browser-based systems to local servers to the cloud). Each transition introduces latency and cost, yet users' expectations for interactivity remain unchanged.
- **Design Dependence:** Interface design decisions directly affect system optimizations, and vice versa. For instance, replacing buttons with sliders or adding new filters can invalidate prior precomputation strategies. This tight coupling forces designers to understand both systems optimization and interface design.

The Data Agents and Processes Lab is exploring three classes of questions:

1. What system primitives should new systems provide?
2. How can we make it easier and more scalable to design new data interfaces?
3. What new capabilities should next-generation interfaces support?

2 Synthesizing Interactive Data Interfaces

2.1 Project Overview

The project "Precision Interfaces" explores how to automatically synthesize interactive data interfaces from example queries.

2.2 Problem Context

Building data interfaces requires both SQL/analytics skills and design expertise. Existing tools (e.g., Tableau, Metabase) make rigid assumptions and cannot handle complex queries. Most scientists can perform analysis but lack the time or training to turn it into accessible applications.

2.3 Goal

The goal is effectively turning a set of example queries into a structured, interactive system.

2.4 Core Philosophy

A data interface consists of two main components:

1. A module that renders query results visually.
2. A set of interactions that allow users to modify the underlying queries in predictable, structured ways.

The total range of possible user interactions defines the expressiveness of the interface, while Precision Interfaces does the opposite. It starts from a few sample queries and reconstructs the hidden interface that could have generated them.

2.5 Technical Approach

Queries are modeled as parse trees (e.g., A=1, B=10). Based on these structures, the system performs several key steps to translate structural patterns in the queries into interactive interface elements.

- Introduce choice operations, allowing users to switch between query branches through interactive components (buttons, dropdowns, sliders).
- Apply transformations to generalize specific query elements (e.g., fixed values $\rightarrow$ ranges, categorical domains).

Construct interfaces via a three-step mapping process:

1. Map query results to visual representations.
2. Map choice nodes to user interactions.
3. Map tree structure to layout configuration.

Finally, candidate interfaces are evaluated through cost functions (e.g., Fitts's Law, GOMS), and a search-based optimization algorithm (MCTS) is used to explore design alternatives and select the most effective interface configuration.

2.6 Intermediate Representation

The system operates on an intermediate representation (IR), a structured layer that connects user intentions with interface generation. This IR can be derived from multiple sources, including:

- historical query logs
- interactive notebooks
- natural language inputs

It captures the choices and variations implied by these inputs (for example, filters, groupings, or value selections) and expresses them in a form that the system can systematically analyze. The IR is represented as an extended Parsing Expression Grammar (PEG), which defines the rules and constraints for what constitutes a valid, expressive, and interpretable interface. This grammar-based structure allows the system to reason about different interface possibilities, verify their correctness, and generalize across diverse types of analytical workflows.

2.7 Applications & Extensions

The intermediate representation (IR) framework enables several practical extensions, which include:

- Automatically generate tutorials that show users how to interact with newly created interfaces.
- Support interface optimization that helps the system adjust layouts or interactions for better usability and performance.

The team further developed the Physical Visualization Designer (PVD), a system that connects interface design goals (such as desired response time or latency) with available system resources (like browser memory or server capacity).

3 Demonstration

To conclude this part of the talk, the guest lecturer presented a live demonstration illustrating how interactive interfaces can be synthesized automatically from example queries.

3.1 Interactive Interface Synthesis in Action

The demo was based on a COVID-19 database. The system allowed the user to load the database, execute analytical queries, and visualize their outputs. For instance, the lecturer issued a query that computed the daily sum of COVID cases and rendered a corresponding visualization.

From this single query, the system provided a button labeled *Generate Interface.* Clicking the button synthesized an initial interactive interface. At this stage, the result resembled a conventional visualization recommendation system: it created a straightforward chart based solely on the current query.

However, the power of the system emerges when multiple related queries are executed. The lecturer demonstrated additional queries—such as breaking down case counts by state or restricting the analysis to specific time windows. Users (or automated agents) can select these example queries, and the system synthesizes a more expressive interface that unifies the interaction patterns implied by the examples.

The synthesized interface allowed users to:

- Select specific states (e.g., California, New York, Alaska).
- Toggle between time ranges such as the last 30 days or last 7 days.
- Combine both state selection and time-range adjustments.

Crucially, all of these controls were automatically inferred from the structural variations found in the example queries.

3.2 From Natural Language to Interfaces

The lecturer further demonstrated that the system can synthesize interfaces even from natural language prompts. For instance, when asked: *"What about the total COVID cases or deaths across all the states in the U.S.?"*, the natural language question was internally translated into a set of candidate queries. Those queries were then passed into the same synthesis pipeline.

The resulting interface included a heat map of the United States. Clicking on a state in the heat map was semantically equivalent to selecting that state through an interaction widget (e.g., a button). The system also generated controls allowing users to switch between visualizing cases or deaths. Compared to earlier examples, this visualization both generalized the previous synthesized interface and supported cross-visualization interactions.

This demo highlighted the broader goal: identifying the *simplest* interactive interface that can express the full diversity of user intentions captured by either example queries or natural language inputs.

3.3 Discussion: How the Interface Is Generated

Not LLM-Based. The lecturer emphasized that the interface generation process does **not** rely on LLMs. Instead, it uses the search-based synthesis procedure introduced earlier in the lecture:

1. Convert input examples into the intermediate representation (IR).
2. Perform search over possible interface designs using the IR.
3. Evaluate candidate designs using interface cost models.

LLMs alone do not perform well at generating complex interface specifications or executable visualization code. However, LLMs can still be helpful—particularly for mapping natural language descriptions into the IR. The lecturer noted that with a few-shot prompt, an LLM can generate IR structures reasonably well.

Design Choices and Cost Models. Another question concerned how the system decides between different interface components—e.g., using a map vs. a dropdown menu vs. a set of radio buttons.

The lecturer explained that the system optimizes for usability using established cost models such as:

- **Fitts's Law** (predicting time to acquire a target)

- **GOMS** (modeling cognitive effort for sequences of operations)

The system scores different interface candidates based on how easy it would be for a user to perform the analysis sequence implied by the example queries. For instance, choosing a state directly on a map often requires fewer steps and less cognitive load than selecting from a long dropdown list; bar charts with many states are also visually overcrowded. Thus, for this specific demo, the map-based interaction received a lower cost under the usability model.

The lecturer stressed that:

- The cost function is not perfect; it can be arbitrarily complex.
- User preferences or historical interaction patterns can be incorporated.
- The system is designed to allow designers to inject their own preference models.

Overall, the system treats interface generation as a formal search problem with pluggable cost functions, rather than making subjective design choices on its own.

4 Expanding Interaction Models

The lecturer next discussed a broader question: even if we can synthesize interfaces at scale, perhaps today's interaction models themselves are too limited. Modern analytical tools support only a small set of interaction mechanisms, and expanding these mechanisms may unlock more expressive data interfaces.

4.1 Limitations of Current Interaction Models

Although data systems have evolved significantly, the fundamental ways in which users interact with analytical interfaces remain narrow. The lecturer summarized three dominant classes of interactions available in most commercial tools today:

1. **Creating new charts.** Users specify how data should be visualized—typically by dragging attributes onto axes (as in Tableau and other dashboard builders). Under the hood, such operations translate into SQL queries such as `GROUP BY` with aggregate functions.
2. **Manipulating chart contents.** Users adjust filters, sliders, or predicates directly on a visualization. For example, moving an age slider changes the corresponding predicate in the underlying query.
3. **Coordination between charts.** Selections in one visualization automatically constrain or highlight data in another. For example, brushing an age range in one view may update predicates in a related view. This form of interaction is widely used in multi-view visual analytics systems.

However, these three categories exclude a surprisingly important user need: **comparison**. Despite being central to analytical reasoning, comparison did not exist as a formal interaction class until very recently.

4.2 Motivation: The Need for Comparison

Modern interfaces place a heavy burden on the user to visually estimate differences or manually compute comparisons. For instance, in a financial dashboard, a user may want to:

- Compare Salesforce vs. IBM stock performance over time.
- Compare today's price vs. a stock's historical trend.
- Compare JP Morgan's movement with IBM's.

Current tools offer no direct mechanism for such comparisons. Users must eyeball charts, manually align values, or write custom queries or scripts.

To address this, the research group introduced the concept of **View Composition Algebra**, a new class of interaction designed specifically to support comparison operations in data interfaces.

4.3 View Composition Algebra

View Composition Algebra extends traditional interaction models by introducing comparison as a *fourth* class of interaction. It allows users to directly combine, align, and compare visualizations through intuitive gestures, rather than coding or mentally estimating differences.

4.3.1 Example Interactions

The lecturer demonstrated several examples:

Computing Differences. A user may drag two line charts (e.g., Amazon and Google stock prices) together and request to "see the difference." The system:

1. Aligns the dates between the two time series.
2. Computes the pointwise difference.
3. Renders the resulting derived chart.

Comparing With Historical Self. A user may compare Amazon's current prices with its own historical values. This operation requires an *outer join* because the historical and current datasets may not align perfectly.

Superposition. A user may request to "overlay" Amazon and Google. In this case, the system takes the union of the datasets and adds a new attribute (e.g., stock ticker) to distinguish them in the combined visualization.

These examples demonstrate that enabling comparison requires a principled definition of semantics, including alignment rules, join types, and attribute handling.

4.4 Challenges: Comparability and Semantics

Several open questions arise when building such a system:

- **When are two datasets comparable?** For example, comparing AutoZone's stock price to a non-financial metric is nonsensical. The system must detect and prevent invalid comparisons.

- **What are the composition semantics?** Depending on the operation (difference, overlay, union), the system must choose the correct alignment strategy and data transformation (e.g., inner join vs. outer join, aggregation rules, attribute augmentation).
- **How can these semantics generalize to any chart type?** Users may want to compare bar charts, scatter plots, multi-view dashboards, or combinations thereof.

View Composition Algebra provides a structured way to reason about such operations and support comparison as a first-class interaction in analytical systems.

5 The Scorpion System

Professor Wu introduced another capability absent from most interfaces: comparing with expectations. Scorpion addresses expectation-driven anomaly explanation. Kay's "Same Stats, Different Graphs" demonstrates visualization's capacity to reveal patterns obscured by summary statistics. Scorpion enables systematic comparison between observed and expected data distributions.

5.1 Explaining Anomalies

The explanation workflow comprises:

1. Data collection and visualization
2. Anomaly identification
3. Causal predicate discovery

Given sensor temperature data exhibiting distribution anomalies (unexpected standard deviation jumps), Scorpion identifies explanatory predicates (e.g., low-voltage sensors with failing batteries generating spurious measurements).

5.2 Demonstration

Using the sensor dataset, Professor Wu computed average temperature and standard deviation over time. He highlighted anomalous regions and specified expected distribution patterns. When he graduated in his PhD defense, the algorithm took a minute to run; with recent work, the bottleneck is now network latency rather than computation.

The system searches all combinations of predicates in the database. For each predicate, it evaluates: if data under that predicate regime is removed, does the output visualization match user expectations? Top results (e.g., mote ID 18) identify problematic sensors. If removed, the visualization matches expectations.

This requires efficient systems work for computing explanations and defining the problem class formally.

6 Data Lineage for Interactive Interfaces

Lineage defines mappings between screen elements and underlying data records. Explanation queries ("why does this visualization appear thus?") and coordination interactions constitute lineage functions. This section formalizes lineage concepts and enabling system architectures.

6.1 What is Fine-Grain Lineage?

Consider a query joining tables on color attributes, grouping by ID, computing `SUM(quantity * cost)`. Fine-grain lineage tracks record-level dependencies:

- Output record 1 derives from intermediate join results 1 and 2
- Join result 1 derives from first input records of both base tables

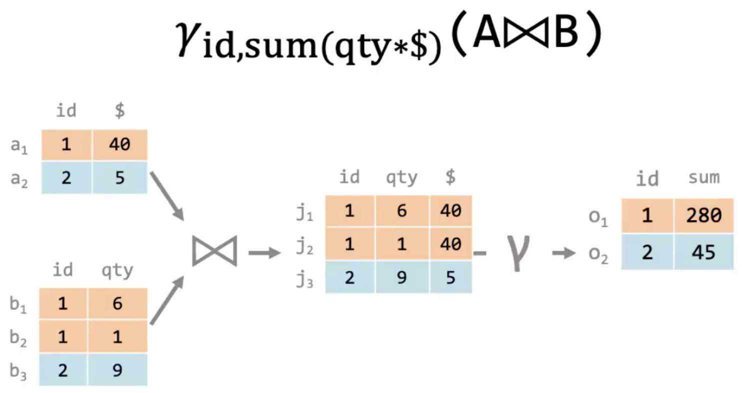

Figure 1: Fine-grain lineage graph showing record-level dependencies through join and aggregation operations. Arrows indicate which input records contribute to each output record.

These dependencies form directed acyclic graphs (DAGs) representing per-operator input-output mappings.

6.2 The Performance Challenge

Lineage edge cardinality scales with query complexity (operator count), potentially exceeding input data size. Prior state-of-the-art systems (Perm, GProM) exhibit 100-1000×+ overhead on TPC-H benchmarks, rendering them impractical for interactive workloads.

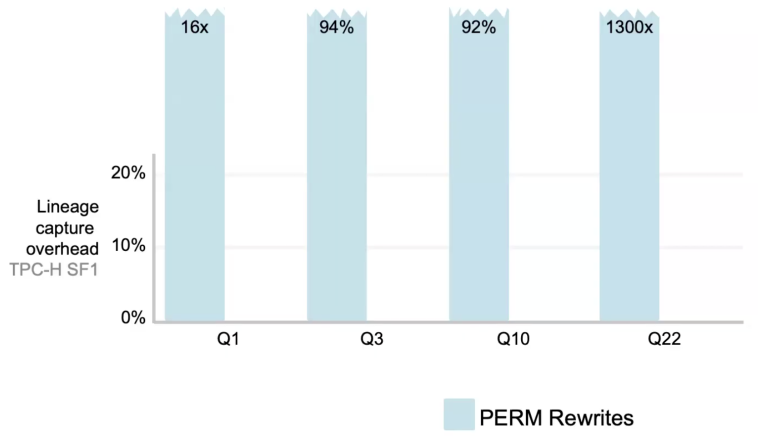

Figure 2: Performance overhead comparison: Prior systems (Perm/GProM) show 100-1000×+ slowdown, while SmokeDuck achieves 0-10% overhead by reusing selection vectors.

Analytical query performance should remain hardware-limited; materializing lineage metadata induces prohibitive overhead.

6.3 Lineage Matters for Interfaces

Explanations: Explanation queries require lineage-based computation over contributing records.

Coordination: Selection operations in one visualization logically select corresponding lineage, enabling cross-chart updates and relative computations.

Code integration: Selected data subsets feed into downstream queries or statistical models.

6.4 The Key Insight

Professor Wu showed a simplified filter operation in an analytical system like DuckDB. In columnar/analytical engines, data is stored column-wise for vectorization. Consider filtering `ATR = 1`:

1. Allocate an index array for matching cells
2. Evaluate equality predicate
3. Store matching indexes (e.g., [0, 2] for rows 0 and 2)
4. Slice output from input using this *selection vector*

Selection vectors precisely encode lineage, indicating which input records each output depends on. Analytical engines already compute intermediate variables encoding lineage as part of query processing.

Because data movement dominates overhead in analytical engines. Data movement at the hardware level requires specifying source and destination, and this information *is* lineage.

6.5 SmokeDuck System Results

Building this insight into a system yields single-digit percentage slowdowns (0-10%) rather than 100-1000×+ overhead. This enables building interactions using lineage for nearly free.

6.6 Interactive Visualization Performance

Professor Wu demonstrated using a forest fire dataset (5 million records) with multiple histograms.

In the baseline system, interactions required re-running queries with updated filters. With SmokeDuck, the system constructs lineage information during initial query execution, using lineage to identify highlighted inputs and update output charts.

The interface loads just as fast initially, but interactions become 100-1000× faster by leveraging lineage rather than recomputation or precomputation strategies.

6.7 Additional Applications

The Scorpion explanation interface shown earlier also uses lineage captured by SmokeDuck, evaluating predicates at a rate of one million per second. At this rate, most explanation queries return immediately.

6.8 Q&A: Caching vs Lineage

When asked whether to cache materialized results or lineage graphs, Professor Wu emphasized it depends on the application and interaction patterns. Optimization strategy selection (materialized result caching vs lineage graphs) depends on application-specific interaction patterns, motivating Physical Visualization Designer's optimization layer. Lineage applicability varies by interaction class, requiring principled trade-off analysis.

For operations with deterministic output locations (e.g., pivot), lineage compresses to simple functions (coordinate transformations). Data-dependent operations require per-record lineage tracking. Established theory exists for many second-order operations.

6.9 Future Importance: Agent Automation

Professor Wu argued lineage is becoming increasingly important for reasoning about automation and agent behavior. Lineage assumes critical importance for agent automation systems. Access control mechanisms prove insufficient for agent-database interactions: agents require data access, yet specific data flows violate regulatory policies (e.g., raw vs aggregate data exposure). Lineage-based policy specifications define permissible data flow structures, enabling enforcement mechanisms that capture actual computational behavior rather than static permissions.

7 Broader Implications and Future Directions

7.1 Agent-Generated Queries as the New Default

Professor Wu notes an emerging shift in real-world deployments: an increasing share of queries submitted to modern data systems originate from AI agents rather than human users. When agents begin to dominate the workload, the nature of query behavior changes substantially. Many of these queries arise from speculative search, multi-step planning, or exploration within a larger task context, meaning that the value of each individual query is not uniform. This suggests that future systems must reason jointly about an agent's task-level planning and the underlying resource management, rather than treating them as independent layers. Understanding how each query contributes to an agent's overall objective becomes essential for scheduling, prioritization, and system efficiency.

7.2 Regulatory Compliance and the Risk of Silent Violations

A second implication concerns compliance and safety. Professor Wu points out that current regulatory constraints—such as laws requiring disaggregated statistics for certain student populations—are often enforced manually by human analysts who remember to check these requirements before releasing results. In contrast, AI agents acting autonomously may easily generate analyses that violate such policies, even if the regulatory text is included in their context. The rate of policy-violating outputs is therefore likely to be extremely high. As a result, systems must provide explicit mechanisms to verify and enforce compliance, rather than assuming that agents will reliably follow the rules. Without such safeguards, the correctness of outputs becomes fragile, and accountability becomes unclear.

7.3 System Vulnerability, Cost Control, and Management Layer Rethinking

Agent-generated workloads also raise concerns about system robustness. According to Professor Wu, agents can unintentionally cripple a system by issuing queries that are performance-heavy, logically harmful, or prohibitively expensive. For example, a single query might trigger a downstream language model invocation costing thousands of dollars. As agents scale, the likelihood of harmful or excessively expensive queries increases dramatically. This situation exposes new forms of vulnerability: performance degradation, incorrect or unsafe outputs, and uncontrolled resource consumption.

These challenges motivate a rethinking of the system management layer. Policies may need to be enforced at the level of data flow, allowing the system to detect and terminate queries exhibiting risky patterns. Managing cost, ensuring correctness, and protecting performance all become central concerns in an environment dominated by autonomous agents. Professor Wu emphasizes that this management layer will likely require more redesign than any other part of the stack as agent-driven workloads continue to grow.

8 Conclusion

Professor Wu's research illustrates that advancing interactive data systems requires rethinking fundamental system primitives as well as the interfaces built on top of them. At the systems layer, familiar assumptions often fail: prefetching can increase latency rather than reduce it, and standard communication protocols may not preserve the responsiveness users expect. This motivates new execution models and new uses of lineage information that support both expressive applications and more effective optimization.

At large scale, another challenge emerges: analysts may need to perform interactive analytics over millions of tables. Professor Wu's work examines how system architectures must change to make such analysis feasible in real time. At the interface layer, the difficulty lies in enabling designers to express complex data-level semantics through user-facing interactions. The comparison interface shown earlier is one example, but many other data-driven interactions require similar rethinking.

At the highest level, Professor Wu's work highlights a deeper principle. Interaction is tightly coupled to the programs executed in response to user actions. Instead of restricting designers to existing query templates, it becomes possible to reverse the direction: identify the types of operations and query transformations desired, and then develop interaction designs that match those capabilities.

This line of work, developed over nearly a decade with contributions from many students, continues to grow as new settings emerge. The same questions now arise not only for human users but also for AI agents that execute tasks on behalf of users. Together, these efforts demonstrate that meaningful progress requires coordinated thinking across system primitives, data models, and interaction design, and that many promising problems remain open for future exploration.