Physical data design is the problem of identifying the appropriate physical layout of data structures for a given application and its workload. Well-designed physical layouts have an enormous impact on the ability of visualization application to run at interactive speed. This problem has been well-studied in other domains, notably the selection of indexes and materialized views for SQL databases pioneered by the AutoAdmin project. Here we outline the key differences between designing physical layouts in the visualization domain as compared to other state-of-the-art physical designers, including AutoAdmin.

1 Unifying Language

1.1 Overview

- Most existing querying and visual languages offer no (SQL, MDX) or limited visual language (immens/falcon/kyrix)
- VizQL (tableau) is the exception and the standard for expressive visual languages
- users specify data source and visual layout and intended data transformations are inferred from visual layout
- no explicit 1:1 correlation between data transformation and visual component. **this is due to a limitation of the compilation process, in which data is partitioned based on the user-specified visual layout that the database can neither access nor concisely express in a single query**.
- absent 1:1 correlation, performance debugging is complex bc there is no unique set of data transformations necessary for any given visual component (tableau experiment)
- we encourage developers to explicitly state the intended data transformation for each given visual component with difftrees as a concise mechanism, providing a 1:1 correlation for performance debugging/optimization

1.2 Background

Visualization designers have two extremes for creating applications. The first is querying languages such as SQL and MDX that give them fine-grained control over the data transformations with little to no support for translating those queries into visualizations, let alone interactions. The other choice is visualization frameworks such as imMens and Kyrix that allow developers to focus on the visualization and interactions but limit developers to the subset of those languages that their optimizations support. *per Eugene: the system needs to guarantee a universal quantifier (for all queries q the interface can express, q will be fast), they need limited expressiveness. The easiest way is to reduce the language expressiveness, but clear that's not the only way.*

Closest to bridging the gap between data and visual languages is VizQL. VizQL provides a rich language for describing tables, charts, and interactions which are then compiled into SQL (or in the case of tables, MDX) queries. This compilation is a multi-step process which iteratively infers the intended set of data transformations.

1.3 Problem

Applications created in VizQL, while well-suited for small datasets, are painful to debug when they exhibit poor performance. Users are cautioned that "poor design causes poor performance"; however, identifying which design element of the application is causing poor performance, and equally as importantly, how to fix it, is not straightforward.

For example, developers can examine the queries that VizQL issues to the database. However, the query only represents a partial step of the data transformation pipeline. In order to optimize performance manually, developers must infer or gain access to the data transformations happening within the application. Only then can developers piece together the full data transformation pipeline, identify bottlenecks, and reshuffle the pipeline by, e.g. pushing slow operations to the data source level rather than the application level.

Ultimately, this process forces developers to reason about the full data-to-visualization transformation anyway; **we argue that encouraging developers to explicitly write down the queries in their application is a more sustainable approach to building scalable visualization applications.** While perhaps requiring more upfront investment, this paradigm allows the application to automate the complex and messy downstream optimization because design elements are directly tied to their performance.

A key challenge here is that visualization applications can easily generate on the order of millions of queries, some of which are well suited to SQL but a significant and important subclass of which are more efficiently represented in MDX or impossible to write without user-defined functions. The brute force approach would force users to script every single query in their interface and update those scripts with corresponding interface changes. This is tedious, repetitive and unnecessary.

1.4 Solution

We propose a language for defining interactive visualizations based on query differencing, in which users write down an initial query and indicate how interactive components modify it. PVD uses an interaction based representation of the query workload. Each visual component is represented as initial query tree, and interactions specify a difftree that encodes how it modifies the query tree and the range of possible values it can take.

There are several advantages of this model. First, a single difftree definition represents a large number of queries without limiting the nature of the workload. Second, it is flexible enough that users are not limited to any single style of querying or language subset. Finally, it serves the critical purpose of untangling the poor design/poor performance problem; slow interactions or slow loads can be diagnosed automatically by the framework by analyzing the individual difftree performance (and the interaction and view it defines).

1.5 Examples

2 Identifying Applicable Optimizations

2.1 Overview

- Existing physical design algorithms make the assumption that they are limited to variations of a small number of data structures (1-2 at most) which they are configuring **why?**
- a small number of grammatical features of querying languages indicate relevant information for configuring the data structure, e.g. indexes speed up filtering

and WHERE or ON clauses indicate filters, projection and aggregation indicate dimension/measures for MDX/cubes

- for optimized frameworks, can only specify visual layouts that are supported by the core optimization in framework so no need to include info about visual layout in optimizing logic
- because PVD supports a variety of optimizations whose relevance depends on information spanning the query workload, data source, and visual layout need to (1) identify what new info from visual layout/data source and (2) which grammatical features of our input language impact physical design
- we show that various application level and language features directly correspond to or supplant inputs to known physical design algos as well as empirically show what new info has meaningful impact

2.2 Background

2.3 Problem

Existing PD algorithms embed relevant information (e.g. selectivity for SQL, density for MDX) in the workload definition that is the key input for the design problem. It follows that in a new domain, we can expand or change the encoded information to meet the application needs. However, determining which information is relevant is nontrivial. The extreme solution is to throw all of the application and workload information into the design problem, but even then it is still unclear which information is useful. Instead, relevancy is demonstrated through empirical and logical analysis of which factors impact the resulting design.

2.4 Solution

PVD includes auxiliary information about workload, providing empirical evidence for how and why that information is necessary and sufficient for identifying an optimal physical design. In particular, we clearly define which aspects of the front-end design impact performance. PVD demonstrates that key application features directly supplants inputs to known physical design problem (e.g. interaction cardinality supplants the notion of database selectivity) as well as how new information (sequencing and interactivity rates) provide domain specific benefits.

2.5 Example

3 Multi-location query planning

3.1 Overview

- Query planning and query optimizers typically assume that (1) physical layouts are static and (2) queries will run in the same physical space as the data
- the latter assumption is often violated in distributed systems, in which federated query planning has shown that traditional query planning techniques yield suboptimal results in distributed environments

- Visual applications violate both of these assumptions (clients refresh data from server, queries are run in client/server/database)
- we show that both distributed/non-distributed query planning techniques are also suboptimal if both assumptions are violated and provide an alternative technique

3.2 Background

Existing physical design algorithms also assume that workloads will run in the same physical space as the physical layout they design and that the physical layout will be largely static. By contrast, visualization applications typically require a client-server architecture in which a (dynamic) portion of data is located on the client and a (relatively static) portion of data is located on a server and/or database.

3.3 Problem

An optimal physical layout needs to effectively utilize resources located on the client, a task for which existing algorithms are unsuited for two reasons. First, it benefits from auxiliary information about the application and user preferences. Second, even given that information, optimizing a plan for partitioned execution is fundamentally at odds with the goals of traditional query planning, resulting in optimizing rules that directly conflict with each other.

3.4 Example

4 Related Work