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 Background

Visualization designers have two extremes for creating applications. The first is querying languages such as SQL and MD 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 a subset of those languages.

Closest to bridging this gap is VizQL, in which users are given 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 three-step process where no single query maps directly to a visual component.

1.2 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. In this scenario, developers are set back to square one, manually troubleshooting through a flowchart of possible causes whose suggested solutions often end in ""consult a data engineer"".

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.

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.

1.3 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).

1.4 Examples

2 Identifying Applicable Optimizations

2.1 Background

A shared characteristic of existing physical design algorithms is that they operate over well-defined input languages that implicitly limit the expressible workload, and hence, limit the suite of relevant optimizations. For example, DBMS accepts SQL workloads which are naturally suited to indexes and materialized views, while OLAP workloads are multi-dimensional expression (MDX) queries that naturally define a datacube. Additionally, visualization specific optimizations such as location based indexes (kyrix), dense datacubes (imMens), or view-level tiling (Falcon) all represent additional optimizations choices whose relative benefit depends on the current application design.

2.2 Problem

An advantage of constraining the input language (SQL, MDX) or the space of interactions (kyrix, immens, falcon) is that the workload gives natural hints for appropriate physical design. Consequently, a general input language, while useful for iterating through multiple designs in the same framework, inherently loses the notion of *applicability* , e.g. which data structure is appropriate for the workload?

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.

2.3 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.

2.4 Example

3 Client-Server Architecture

3.1 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.

3.2 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.

3.3 Example

4 Related Work