Explaining Data in Visual Analytic Systems

by

Eugene Wu

B.S., University of California, Berkeley (2007)
M.S., Massachusetts Institute of Technology (2010)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Author .....
Department of
Electrical Engineering and Computer Science
December 18, 2014

Certified by .....
Samuel Madden
Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Explaining Data in Visual Analytic Systems

by

Eugene Wu

Submitted to the Department of
Electrical Engineering and Computer Science
on December 18, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

ABSTRACT

Data-driven decision making and data analysis has grown in both importance and availability in the past decade, and has seen increasing acceptance in the broader population. Visual tools are needed to help non-technical users explore and make sense of their datasets. However even with existing tools, many common data analysis tasks are still performed using manual, error-prone methods, or simply inaccessible due to non-intuitive interfaces.

In this thesis, we addressed a common data analysis task that is ill-served by existing visual analytical tools. Specifically, although visualization tools are well suited to identify patterns in datasets, they do not help users characterize surprising trends or outliers in the visualization and leave that task to the user. We explored the necessary techniques so users can visually explore datasets, specify outliers in the resulting visualizations, and produce explanations that help explain the systematic sources of the outlier values.

To this end, we developed three systems: DBWipes, a browser-based visual exploration tool; Scorpion, a set of algorithms that describes the subset of an outlier's input records that "explain away" the anomalous value; and SubZero, a system to track and retrieve the input records that contributed to output records of a complex workflow. From our experiences, we found that existing visual analysis system designs leave a number of program analysis, performance, and functionalities on the table, and proposed an initial design of a data visualization management system (DVMS) that unifies data processing and visualization and can help address these existing issues.

Thesis Supervisor: Samuel Madden
Title: Professor

ACKNOWLEDGMENTS

We stand on the shoulders of giants
yet innumerable people lift us onto those shoulders.

The big ones: Sam Madden has been the most consistent and positive source of ideas, perspective, freedom, encouragement, cheerleadering, laughter and funding... for chocolate. His firm grasp on what really matters in both research and in life has been a constant source of of inspiration as I grew as a researcher and human, and his deep baritone voice has been a constant source of envy. From him, I learned to ask the question "what's *cool*?". Michael Stonebraker has been un-yieldingly honest, insightful, and supportive. From him, I learned to always ask "what's *useful*?". From Nickolai Zeldavich I learned the value of, if not the implementation of, a strong work ethic. He graciously passed me during my qualifying exams and *still* agreed to be on my Ph.D. committee.

This work couldn't have been possible without help from Dr. James Michaelson's lab, Martin Spott and researchers at British Telecom, and my user study participants.

The graduate experience is neither complete nor possible without the enormous intellectual and emotional support from my colleagues and friends. Philippe Cudre-Mauroux mentored me early on and is an excellent friend with a wonderful karaoke voice. Alvin Cheung has an enormous mental repository and taught me to thoroughly question the fundamentals. Lenin Ravindranath is the fastest ideas-to-prototype-to-publication research I know, and I learned a lot from watching him shape ideas. Carlo Curino is the warmest, silliest Italian I know, and showed me how to balance goofiness and research precision. Adam Marcus both introduced and let me join him on his journey through the database crowdsourcing world. Beyond being an amazing friend, he is also the moral guidepost upon which I measure my ethical decisions.

Sam provided the funds, but Sheila Marian made sure I got those resources.

I could not have been part of a better research group than the MIT Database group – thank you all. There were no better officemates than the illustrous members of the G930 office – yuan mei, akcheung, ravi, pcm, yonch, asfan, and nirmesh.

I owe gratitude to so many who have helped make Cambridge my second home: Anant Bhardwaj, Ramesh and Priya Chandra, Alvin Cheung, Jenny Cheung, Austin Clements, James Cowling was first to welcome me to MIT, Neha Crosby, Cody Cutler, Aaron and Emily Elmore, Gartheeban, Michal Depa, Edward, Grace and Everest Benson, Carlo and Christy Curino, Irene Fan, Ben Holmes, Evan "<3 transactions" Jones, Neha Narula, Ravi Netravali, Karen and Bryan Ng, Aditya Parameswaran, Jonathan, Noa and Ayala Perry, Raluca Popa, Irene and Dan Ports, Asfandyar Querishi, Lenin Ravindranath, Meelap Shah, Lynn and Edward Sung, Jen and Terence Ta, Stephen Tu, Tosci's, Grace Woo and Szymon Jakubczak with whom I have shared a home, a mortgage, and my birthday, Yang "big man" Zhang and Christine Rha, Yuan Mei, Richard Zhang.

I would never have discovered the supportive and inclusive database community if not for the many many people at UC Berkeley. Shawn Jeffery and Shariq Rizvi saved me from a directionless first summer and pulled me into my first foray in the database group. Michael Franklin and Joeseph Hellerstein, to this day, provide continued support for a precocious kid who once thought he knew everything. Yanlei Diao showed me how to mold a simple class project into my first and still most cited "real" paper. Before joining MIT, Mr. Jeffery convinced me to play at Google, where Alon Halevy and Michael Cafarella introduced me to research at scale.

So many thanks to Lydia "Zhenya" Gu, who has stuck with me despite all of my wonderful qualit... I mean faults and strangeness.

I would have nothing if not for my parents and my brother Johnny Wu.

Contents

Figures and tables

1 Introduction

Analyzing data is an exploratory process, where the analyst attempts to understand the trends and patterns hidden in the data. Technology trends have continued to change the nature of data analysis in two seemingly opposing directions. On one hand, datasets that are gathered from an increasing number of sources, such as financial markets, sensor deployments, and network monitoring, are also growing in size, dimensionality, and complexity. On the other hand, the lower costs and increasing accessibilty to acquire, store, and process data is broadening the class of data analysts to include more and more non-professional and novice programmers. These trends point to the need for systems that are both easy to use for a broad range of data users, and can effectively support the user's exploration process, even when working with large and diverse datasets.

In recent years, there has been significant progress in interactive visualzation systems, such as Polaris [102] and Tableau, that simplify how analysts interact with databases. These systems translate direct manipulation operations, such as mouse clicks and dragging operations, into database queries and visualization operations. This allows analysts that are not familiar with query and programming languages rapidly explore many views of the data with minimal training.

1.1 EXAMPLE

During the user's exploration process, some visualizations will reveal surprising patterns that the user will want to understand. For instance, a sales analyst that is monitoring daily sales transaction data may be surprised by a sudden spike in recent sales revenue. This increase could be due to a multitude of reasons – the company's expansion into a new market that triggered sales from new users, an increase in popularity within a specific user segment, or simply an accounting error that over-estimated some sales amounts by an order of magnitude – none of which are obvious through the visualization. Although it is simple to visually identify the anomalies, it is significantly more difficult to determine the reasons

behind them using existing systems.

A common approach is to look for attribute values (or combinations) that are highly correlated with the anomalies. Analysts will select subsets of the data that mtach different combinations of attribute values and observe how the anomalies in the visualization change. However, visually comparing the visualizations can result in sub-optimal or incorrect conclusions due to the limits of human graphical perception [32] – our ability to decode quantatative information for visual encodings. In addition, the number of possible combinations increases exponentially with the dimensionality of the dataset and quickly dwarfs the number that can be feasibly tested by hand.

While it may be possible for professional data analysts to write programs to automate some of this analysis, this requires switching to a different development environment and writing a separate program for each visualization. In addition, novice users that depend on the application to perform analyses [62] will not have the technical expertise, and resort to a manual process that can only test a small number of combinations.

This highlights a core limitation of existing visual analytics systems – they are designed to display data, but lack facilities to explain the underlying patterns in the context of the visualization. In this dissertation, we explore the mechanisms that enable visual exploration *and explanation* of data. We develop visual interfaces to specify anomalies and present explanations, data-mining algorithms to generate explanations for user specified anomalies in the visualization, and data-processing systems to support these functionalities.

1.2 A SOLUTION SKETCH

Developing a general purpose system that can support this form of *explanatory interaction* depends on the specific visualization that the user creates, how the data was transformed prior to visualization, and the types of anomalies that the user is interested in. Consider the problem above; a solution needs to perform the following steps:

Specify Anomalies

Visualization systems often support a large class of possible visualizations, each encoding data into visual properties in a custom way. Thus, the system needs to provide a uniform way for the user to express anomalies in *any* visualization expressible by the system. For example, in a heat map, the positional attributes matter less than the luminosity or hue of the selected points. In contrast, a typical bar chart will encode the primary variable of interest along the y-axis position, whereas the hue is used for grouping the bars by a

categorical variable. When the user selects a portion of the visualization, it must be easy to specify the precise output and the attributes that are anomalous.

Backwards Lineage

In general, every output point is an *aggregate* that is generated by combining data from multiple input tuples. In order to work backwards from the output to its corresponding inputs (its lineage), both the visualization and database systems need to track lineage information and provide a queryable lineage interface. Although some database systems [113] have been instrumented to track lineage information, few can ensure resource guarantees when processing large datasets. In addition, visualization clients are implemented imperatively, making tracking data lineage through the visualization layer very difficult. Thus, a key challenge is a system design that can automatically, and efficiently, track lineage across both data processing and the visualization layer.

Generate Explanations

For each outlier result, we need to generate a set of possible explanations for its value. In the example above, the explanation is a combination of attribute values that most caused the result to be an outlier. However, it is not clear what a good explanation is, and manual heuristics to this problem often use inconsistent preference criteria. Thus the key challenges are to define a formal definition of a "good explanation" and develop algorithms that can efficiently find them.

Interface Integration

The set of explanations that are generated can potentially be very large, and the visualization needs to include an interface for users to efficiently navigate through the possible explanations and evaluate them by hand. In addition, the explanation process needs to be integrated such that it augments, rather that replaces or disrupts, the user's normal data exploration workflow.

1.3 DISSERTATION CONTRIBUTIONS

This thesis contributes novel systems and algorithms that expand the scope of analyses that analysts can express through a visual interface. The overall architecture and each of the systems is summarized in Figure 1-1. The visualization system translates user interactions, such as clicks and mouse drags, into SQL queries submitted to the database, and updates

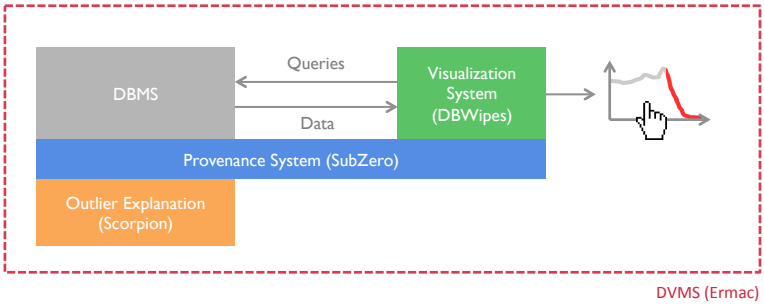


Figure 1-1: Architectural summary of system contributions (colored boxes) in this dissertation.

the visualization with the query results. The provenance system tracks the provenance metadata throughout the query execution and efficiently retrieves the records that were used to generate points and lines in the visualization. The outlier explanation system uses provenance information to generate explanations to outliers that the user finds in the visualization. The above components are designed on top of existing database and visualization architectural designs. In contrast, rather than extending existing systems to support these functionalities, the data visualization management system is a clean-slate design that aims to simplify many of the analysis, performance, and engineering challenges with existing database and visualization system designs. This section describes each of these components in more detail.

High-throughput Provenance System

The overhead of tracking input-output record relationships can be orders of magnitude more resource and time intensive than the baseline execution system without provenance, and existing provenance systems are not well equipped to manage the resource overheads. Chapter 3 presents the design and implementation of SubZero, a provenance management system that extends high-throughput workflow execution systems with the ability efficiently expose provenance metadata and manage the storage and runtime costs of tracking this information. Our experiments on two scientific benchmark applications show that such a management system is necessary in data-intensive environments.

Novel Algorithms for Outlier Explanation

In Chapter 4, we present Scorpion, a system that explains outliers in the result of aggregation queries. Scorpion mines combinations of attribute values (predicates) to find combinations that most *influence* the values of those outliers. Our contributions include a novel sensitivity-based *influence metric* that assesses the amount a predicate contributes ot outlier values for

arbitrary aggregation functions, and efficient algorithms for mining the space of possible predicates for common classes of SQL aggregation functions.

Interactive System for Exploring Data

Chapter 5 introduces DBWipes, a visual analytics tool that is integrated with the Scorpion explanation system. DBWipes contributes an interface for assessing the influence of input data on anomalies in a visualization, and a direct manipulation interface for specifying visualization anomalies and asking *why* those results are anomalous. We present user study results that show Scorpion significantly increases how quickly and accurately users are able to understand anomalies in a visualization, and identify a number of common user mis-perceptions when search for explanations that can lead to incorrect conclusions.

Integrated Data and Visualization Management System

Finally, we use our lessons learned to propose the design of a *Data Visualization Management System* (DVMS) that combines data processing and visualization tasks in a single relational execution engine. The system translates high-level declarative visualization specifications into a relational execution plan that produces as output a visualization. This integration enables provenance information to be tracked from the input data records to the rendered outputs. We describe the system design in Chapter 6, and outline a number of research opportunities that result from an integrated design.

2 A Brief Lineage Primer

Before describing our approach to tracking and query provenance, it is helpful to describe what lineage is and how it is defined in this dissertation. This chapter provides a brief overview of lineage, and defines the general workflow execution model, lineage model, and lineage query model that is used in subsequent chapters. In addition, we will introduce several examples of applications that track and use lineage, and the key dimensions that can be used to classify and compare different lineage systems. Our goal is to motivate the value of lineage information, and provide enough context and formalism so that the subsequent chapters can be understood. Chapter 7 provides a more detailed list of related publications, theses, and surveys for the interested reader.

2.1 PROVENANCE AND LINEAGE BACKGROUND

This section introduces the concepts of provenance and lineage, and comments on their semantics.

2.1.1 PROVENANCE

Provenance was originally described in the context of the art world to describe an art piece's creation and ownership history. Similarly, provenance in computational systems is metadata that fully describes the origins of a data artifact. This can include input data, intermediate results, processing components, arguments, and annotations. Tracking this information is useful for post-hoc debugging or analysis and can answer questions such as *"What files were used to create this result?"* and *"What result files were computed by this buggy implementation?"*.

This metadata can be modeled as a directed acyclic graph (DAG), where an edge $A \rightarrow B$ means that A was used to derive B. The nodes typically refer to a particular version of a process (e.g., operators, scripts, programs), the process arguments (e.g., configuration files,

input arguments) and data files. Each node may have a number of properties, such as a file system path, a version number, or a constant argument value.

Figure 2-1: Provenance of a simple SQL query plan.

For example, a scientist that runs several scripts to generate a graph of his experiment results may want to track the order in which she ran the scripts and which data files she used. In this case, nodes would correspond to the scripts and data files. As another example, database systems translate SQL queries into an operator tree whose leaf operators consume input relations, and the root operator outputs the result relation. In this context, Figure 2-1 illustrates the provenance of the following query:

SELECT $sum(a)$ FROM T WHERE $a > 10$

The provenance consists of the operators in the query plan (black text), the input, intermediate, and output relations (grey text), and the dependency information (edges).

Once the graph has been created, users can query the graph using graph-like query languages such as Lorel [3], SparQL [90], PQL [49] or by writing graph traversal programs.

2.1.2 LINEAGE

A subset of provenance, called *data lineage* is specifically concerned with the dependencies between the data records in the inputs and outputs of a computational process. For example, a visualization system may want to track the relationships between pixels in the rendered image and the data records in the database so that users can select a set of pixels and examine their input data.

Data lineage systems such as Trio [113] and SubZero [118] (Chapter 3) contrast from general provenance systems by the finer granularity in which the data provenance is tracked. Lineage systems typically model nodes in the provenance graph processes and *individual data records*.

This adds two wrinkles towards tracking dependency information. First, tracking fine-grained dependencies is significantly more difficult than coarse file relationships. While it may be easy to instrument the runtime (e.g., the file system [85]) to automatically track and add dependency information to the files that processes read and write, record-level relationships

depend on understanding the semantics of the processes, which may be black-boxes to the runtime.

Second, the quantity of lineage information increases with the size of the datasets. In the worst case, every output record depends on every input record and the number of relationships is quadratic with respect to the dataset size. As datasets increase from hundreds to millions or billions of records, lineage information can easily become the dominant cost in the execution system.

2.1.3 PROVENANCE AND LINEAGE TERMINOLOGY

The distinctions between provenance and lineage can often lead to confusion because the terms tend to take on differing meanings depending on the scientific discipline and context. In some articles, the terms provenance and lineage are used interchangably, whereas in others, lineage is used as a specific subset of provenance that is concerned with data item relationships.

In this dissertation, we use the latter form; *lineage* refers to dependencies between *data* (i.e., edges that connect two data artifacts), whereas *provenance* is concerned with general dependencies between data files, operator execution history, and execution arguments. In addition, we distinguish between *coarse-grained lineage*, which tracks relationships at the dataset granularity, and *fine-grained lineage*, which tracks data record relationships as described in the previous subsection. Unless otherwise specified, *provenance* is concerned with coarse-grained lineage, while *lineage* refers to *fine-grained lineage*.

2.1.4 APPLICATION DEFINED SEMANTICS

The reason we are vague about the exact structure and meaning of the relationship $A \rightarrow B$ is because applications typically define their own semantics. The Open Provenance Model [83] (OPM) is an effort to standardize core provenance concepts. It characterizes high-level notions such as Artifacts such as datasets or files, Processes that consume and produce artifacts, and Agents that execute processes. However, it does not dictate the storage representation, *which* metadata to actually store, nor *how* relationships in the provenance graph are interpreted by a specific application.

One reason for this difficulty is that nearly every discipline and application has different provenance needs: scientists are concerned about reproducibility and want to track their script executions and data files; desktop applications track operation logs to provide history and undo features; security systems track information flow control (provenance) to avoid

leaking sensitive data; auditing systems are interested in a digital paper trail; probabilistic database systems use lineage to compute the uncertainty of computation results.

Each of these applications cares about different types of provenance (e.g., script names vs process arguments vs system calls), varying granularities of lineage information (e.g., data files or data records), and define different notions of correctness (e.g., security systems may not tolerate missing lineage relationships, but false positives may be acceptable).

As a simple example, consider the following simple Python code snippet, where `input` is an array containing cells with two attributes, `type` and `value`, and the code computes the sum of of all valid cell values:

```
sum = 0
for cell in input:
  if cell.type == 'valid':
    sum += cell.value
return sum
```

One possible interpretation is that the output value sum depends on every cell in the input if *any* attribute of the cell (e.g., **cell.type**, **cell.value**) was read in the process of computing sum. In information flow control, this is called the *implicit flow* of the program, which takes into account data used in the program's control structure. Tracking implicit flows is important when the application uses provenance for security purposes.

Alternatively, the developer may only care about *explicit flows*, and define the lineage as all input cells whose **cell.value** was directly used to compute sum's value. This may be sufficient for simple diagnostic use cases.

This example shows that multiple acceptable semantics can be defined for the same operator and the choice ultimately depends on the application that will use the lineage. In this dissertation, our provenance systems are only concerned with providing efficient lineage storage and querying *mechanisms*, and leave it to the applications to define their own semantics.

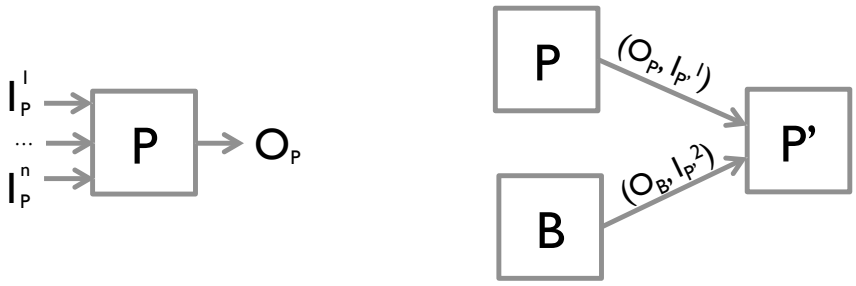
2.2 WORKFLOW DATA AND EXECUTION MODEL

In this section, we formalize what we mean by "dataset" and "workflow".

2.2.1 DATA MODEL

We define a dataset as a collection of records where the records in the collection adhere to a consistent schema, each record consists of values for each attribute in the schema, and there is a unique identifier for each record. For instance, records (or cells) in a matrix or array are identified by their array coordinates, while records in a database relation are identified by the values of their primary key attributes.

2.2.2 EXECUTION MODEL



(a) Input/Output for a single operator.

(b) Edges between three operators.

Many systems, such as Hadoop, business processes, database systems, model execution as a workflow of operators, controlled by a workflow management system. Developers register operators and datasets, connect operators into workflows that the system executions efficiently. For example, databases compile SQL queries into a tree-structured operator workflow.

We assume that the workflow execution system applies a fixed sequence of operators to some set of inputs. Each operator is uniquely defined by an ID and a version number, and operates on one or more input datasets (e.g., tables or arrays), and produces a single output object. Formally, we say an operator P takes as input n objects, $I_P^1, ..., I_P^n$, and outputs a single object, O_P (Figure 2-2a).

Multiple operators are composed together to form a workflow, described by a workflow specification, which is a directed acyclic graph $W = (N, E)$, where N is the set of operators, and $e = (O_P, I_{P'}^i) \in E$ specifies that the output of P forms the *i'th* input to the operator P' (Figure 2-2b).

An instance of W, W_j, executes the workflow on a specific dataset. The workflow is executed in a push-based fashion, where each operator runs when all of its inputs are available.

For simplicity, we assume that workflow systems are "no overwrite," meaning that intermediate results produced as the output of operator execution are always stored persistently and can be referenced. Also, we assume that each update to an object creates a new, persistent version. Previous work [117] has explored which intermediate results to store if there is limited storage space, so we don't deal with it here.

2.3 PROVENANCE DATA AND QUERY MODEL

This section describes the provenance data and query in enough detail to serve as a contrast to the lineage models described in the next section.

2.3.1 PROVENANCE DATA MODEL

We loosely model provenance as a provenance graph with "enough information to re-run a workflow instance and reproduce the same results." For example, consider the workflow instance shown in Figure 2-3. The provenance represents an analagous graph that includes the execution arguments for each operator (boxes), references to each dataset T_x, and the edges that connect the datasets to operator input and output ports.

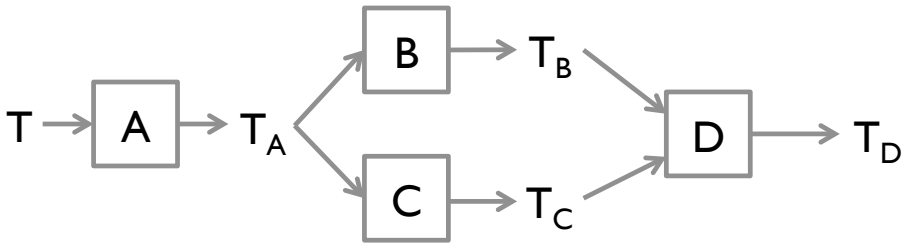


Figure 2-3: Example of a workflow instance. Boxes are operators, each T_x is a dataset, and edges connect datasets to operator inputs or outputs.

In addition, the provenance includes the returns and timings of all non-deterministic calls so that they can be faithfully replayed if an operator is re-executed. This functionality mirrors that present in many workflow systems [21, 67, 103]. Note that data is tracked at the dataset level, so the relationships of individual records are not tracked.

2.3.2 PROVENANCE QUERY MODEL

Provenance queries can be viewed as graph traversal queries over the entire provenance graph. Queries typically fall into three categories: queries agnostic to workflow instances,

queries specific to a workflow instance, and queries that access a specific node in the graph. For example, queries in the former category include:

1. What are all workflow instances that executed operator A?
2. What are all workflow instances that used a corrupt dataset T_i as input?
3. What are all operator instances that computed a result derived from T?
4. What are all datasets that depend on a faulty operator A?

Examples of queries specific to a particular workflow instance W_i include:

1. What are the operators immediately preceeding operator A?
2. What datasets were used as input to operator A?
3. What output datasets depend on input dataset T_i?
4. What input datasets generated output dataset T_o?
5. Find all operator paths between input dataset T_i and output dataset T_o.
6. What input datasets do outputs T_{o1} and T_{o2} share?

Finally, some queries will retrieve metadata about nodes in a specific workflow instance W_i:

1. What were the arguments and recorded non-determinism for operator A in W_i?
2. What is the file referenced by T_x?

The lineage queries in the next section assume the ability to retrive the intermediate datasets of a workflow instance given a path of operators in the provenance graph. Thus, given a path A, B, D in Figure 2-3, the provenance system returns T, T_A, T_B as the inputs to the operators, respectively, and T_D as the output of T_D.

2.4 LINEAGE DATA AND QUERY MODEL

In contrast to the previous section, this section describes how we logically model fine-grained lineage, and the query model that we will use the rest of this dissertation.

2.4.1 LINEAGE DATA MODEL

To support lineage, we assume that each operator has been instrumented with the ability to output lineage information as a side-effect of execution, and that the workflow system has a mechanism to turn this ability on and off. We logically model lineage as a set of pairs of input and output records:

$$\{(out, in) | out \in O_P \wedge in \in \cup_{i \in [1,n]} I_P^i\}$$

Here, $out \in O_P$ means that *out* is a single record contained in the output dataset O_P. *in* refers to a single record in one of the input datasets.

Chapter 3 describes the mechanisms for operator instrumentation, and efficient representations of this lineage information.

2.4.2 LINEAGE QUERY MODEL

Lineage queries are specifically concerned with relationships between one or more sets of records. The queries take as input a set of records, and a path of operators in a workflow, and returns a set of records that constitute the lineage. This formulation can answer questions of the form "what input records do these results depend on?" or "what result records depend on these inputs?"

Users execute a lineage query (the black path) by specifying an initial set of query records C in a starting dataset, and a path of operators $(P_1 \dots P_m)$ to trace through the workflow:

$$R = execute\_query(C, ((P_1, idx_1), ..., (P_m, idx_m)))$$

Here, the indices $(idx_1 \dots idx_m)$ are used to disambiguate the input of a multi-input operator that the query path traverses through.

Depending on the order of operators in the query path, the query is a *backward lineage query* or *forward lineage query*. A backward lineage query defines a path from a descendent operator P_1 that terminates at an ancestor operator, P_m. The output of an operator, P_{i+1} is the idx_i'th input of the previous operator, P_i, and C is a subset of P_1's output dataset, $C \subseteq O_{P_1}$.

A forward lineage query reverses this process, and defines a path from an ancestor operator P_1 to a descendent operator P_m. The output of an operator P_{i-1} is the idx_i'th input of the next operator, P_i. The query records C are a subset of P_1's idx_1'th input array, $C \subseteq I_{P_1}^{idx_1}$. The query results are the records $R \subseteq O_{P_m}$ or $R \subseteq I_{P_m}^{idx_m}$, for forward and backward queries, respectively.

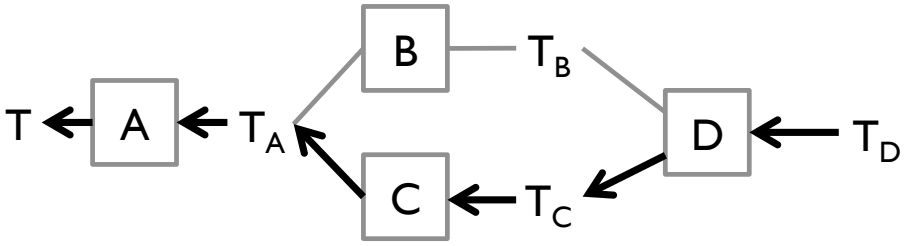


Figure 2-4: Example of a backward lineage query (black arrows)

As a concrete example, the black arrows in Figure 2-4 depicts the path of a backward query $execute\_query(C, ((D, 2), (C, 1), (A, 1)))$. In this query, $C \subseteq T_D$ is a set of result records, $(D, 2)$ distinguishes between D's inputs T_B and T_C and retrieves the input records the second input dataset T_C.

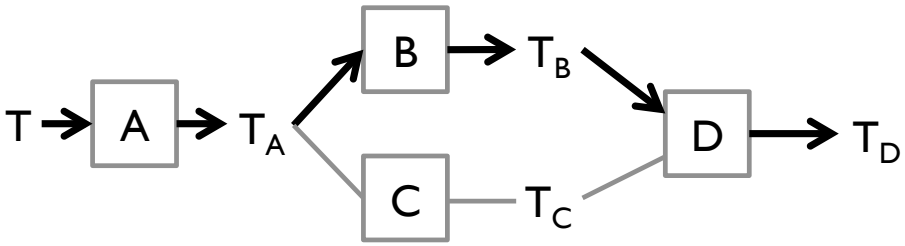


Figure 2-5: Example of a forward lineage query (black arrows)

Figure 2-5 shows the path of the forward query $execute\_query(C, ((A, 1), (B, 1), (D, 1)))$. $C \subseteq T$ is a set of input records in T, and $(A, 1)$ specifies that we are interested in the records in T_A that depend on C when T is used as the first input dataset in A. This distinction is important because the same dataset could be used as multiple inputs to an operator. For example, the values of a matrix M could be doubled by adding the matrix to itself using a binary ADD operator.

There are two reasons why our lineage queries explicitly specify a path of operators. The first is because this disallows ambiguous queries. Consider the query "what records in T generated $C \subseteq T_D$?" for the workflow in Figure 2-3. There are two possible operator paths between T and T_D – A, B, D and A, C, D – and it is not clear how the subsets of T along each of the two paths should be combined. Some applications may use the union, the intersection or arbitrarily pick one of the paths. However, although the semantics are unclear, *execute_query* can be used as a building block to execute these higher-level queries.

The second is because many of the applications we have encountered (described in Section 3.3) want to execute path-based lineage queries. For example, an application may suspect that a specific operator is buggy, and want to inspect its inputs given a set of

anomalous workflow results. The next chapter will describe these applications in more detail, and introduce the SubZero system, which stores, queries, and manages fine-grained lineage metadata for high-throughput workflow applications.

3 High-throughput Lineage

This chapter investigates the design of a lineage management system to support the lineage queries described in the previous chapter for high-throughput data processing systems such as visualization systems. These types of data-analysis applications are quickly moving beyond data *presentation* towards *exploration* and *post-hoc analysis*; it is not sufficient to simply render a static graphic that contains outliers, because users want the ability to e.g., reassess the outlier data, and debug their analyses. Many such functionalities, including the algorithms described in Chapter 4, rely on the ability to query the metadata that identifies how input tuples are related to intermediate and output tuples, or *lineage information*.

Unfortunately, naively tracking these lineage relationships for each intermediate and output record can be very storage and CPU intensive – the storage requirements alone easily scales quadratically with the cardinality of the datasets and linearly with the number of processing steps. The goal of this chapter is to develop a system that can easily incorporate custom analysis operators, and quickly execute lineage queries while satisfying hard application-defined resource constraints.

3.1 INTRODUCTION

Many applications – visualization systems, database query plans, scientific analyses, business processes – are naturally expressed as a workflow comprised of a sequence of operations applied to raw input data to produce an output dataset or visualization. Like database queries, such workflows can be complex, consisting up to hundreds of operations [59] whose parameters or inputs vary between executions.

For example, the Ermac system described in Chapter 6 takes as input a visualization specification that describes the data transformation, layout, and rendering operations, compiles it into a directed-acyclic-graph of relational and custom operators, and executes the operator graph to generate a visualization. When the user finds a surprising data point in Ermac's visualized result, she may want to better understand the source of the result.

At this step, it is helpful to be able to step backward through the processing pipeline to examine how intermediate results changed from one data transformation step to another. If the user finds an erroneous input, she may want to step forward to identify the derived downstream outputs that depend on the erroneous value and possibly correct those results.

This debugging process of stepping backwards and forwards through the processing pipeline extends beyond visualizations. Scientists such as astronomers (cleaning telescope images), genomicists (aligning genomic sequences and cleaning gene expression data), and earth scientists (processing satellite images) all use workflow-based processing systems and want the ability to navigate forward and backward in their pipelines as part of the debugging process [104].

Unfortunately, when the datasets are large, it is infeasible to examine all of the intermediate data at each step, so lineage is helpful to filter the datasets to the subset that actually contributed to the result records that the user is interested in.

3.1.1 CHALLENGES WITH EXISTING APPROACHES

Prior work in data lineage tracking systems has largely been limited to coarse-grained lineage tracking [69, 86], which stores the graph of operator executions and data relationships at the file or relational table level.

On the other hand, systems that track fine-grained lineage either follow an eager or lazy approach. The first, popularized by Trio [113], eagerly materializes metadata about the input data records that each output record depends on and uses this metadata to answer backward lineage queries. The second approach, which we call *black-box*, simply records coarse-grained lineage as the workflow runs, and materializes the lineage at when the user executes a lineage query by re-running relevant operators in a tracing mode. Unfortunately, neither technique is completely sufficient for general workflow applications.

First, applications often make heavy use of user-defined functions (UDFs), whose semantics are opaque to the lineage system. Existing approaches conservatively assume that every output record of a UDF depends on every input record, which limits the utility of a fine-grained lineage system because it tracks a large amount of information without providing any insight into which inputs actually contributed to a given output. This necessitates proper APIs so that UDF designers can expose fine-grained lineage information and operator semantics to the lineage system.

Second, neither the eager nor black-box technique is optimal (with respect to storage costs, runtime overhead, and query performance) are across all workflows. High-throughput workflows can easily consume input datasets with millions of records and generate complex

relationships between groups of input and output records. Eagerly storing lineage can avoid re-running some computationally intensive operators (e.g., an image processing operator that detects a small number of stars in telescope imagery), but needs enormous amounts of storage if every output depends on every input (e.g., an aggregation operation). In the latter case, it may be preferable to recompute the lineage at query time. In addition, applications will often have practical resource limitations and can only dedicate a percentage of their total storage to lineage operations. Ideally, lineage systems would support a hybrid of approaches and take application constraints into account when deciding which operators to store lineage for.

Finally, both techniques are merely two extreme approaches for how to represent and materialize lineage information. Understanding and exploiting the structure between the groups of input and output records will help us develop more efficient lineage representations. For example, suppose an operator adds 1 to each input record. The eager approach would store each output record's corresponding input record. Alternatively, this relationships could be encoded as a function that maps an output record to the corresponding input record with the same primary key, without needing to explicitly materialize any lineage information. There is a need to identify representations are general, simple to express, and efficient.

3.1.2 CONTRIBUTIONS AND CHAPTER ROADMAP

In this chapter, we describe the design of SubZero, a fine-grained lineage tracking and querying system for high-throughput applications. SubZero helps users perform exploratory workflow debugging by executing a series of *data lineage queries* that walk backward to identify the specific input records on which a given output depends and that walk forward to find the outputs that a particular input record influenced. SubZero must manage input to output relationships at a *fine-grained* record level.

SubZero seeks to address the above challenges in the context of scientific applications. We interviewed scientists from several domains to understand their data processing workflows and lineage needs (described in Section 3.3) and used the results to design a science-oriented data lineage system.

In Section 3.5, we introduce a new lineage representation – *Region Lineage* – which exploits locality properties that are prevalent in the scientific operators we encountered. It addresses common relationships between groups of input and output records by storing grouped or summary information rather than individual pairs of input and output records. In addition, it generalizes the existing eager, Trio-style approach.

Alongside the region lineage model, we developed a lineage API that uniformly supports

our new model as well as the black-box approach. Section 3.6 introduces a set of concrete *Region Lineage Representations* that vary from very general and potentially storage intensive, to very efficient but restricted to a special class of operators. Developers decide which representations are optimal for their operator and implement towards the corresponding API.

Each region lineage representation must subsequently be encoded as physical bits and indexed for fast lookups when executing a lineage query. Section 3.7 describes SubZero's various encoding and indexing options and their tradeoffs. Section 3.8 then presents the optimizer that balances these tradeoffs with the user's storage and runtime overhead budgets to pick a globally optimal strategy.

One benefit of separating the lineage data model, the logical representation, and the physical encoding is that the developer only needs to provide as many logical representations as she wishes, and can let the runtime system to pick the best logical representation and physical encoding. This is conceptually reminiscent to the notion of physical data independence in database management systems. This independence property roughly states that physical changes in how the data is stored (e.g., the data format, whether indices are created) does not affect how the data is accessed by the client. This independence is also what allows for query optimization, so that a query optimizer can pick from multiple physical execution plans depending on how the data has been physically stored and its statistical properties. Section 3.9 presents results from our two scientific lineage benchmarks that suggest the *necessity* of an optimizer in a lineage runtime because of the extreme differences between optimal and sub-optimal plans.

3.2 SCIENTIFIC DATA PROCESSING

In this section we introduce the key properties of scientific data processing systems, and provide rationale about why we focus on this class of applications as opposed alternatives such as general database systems or Hadoop-based data processing systems.

3.2.1 SCIENTIFIC WORKFLOW PROPERTIES

Scientific workflows are primarily defined by the types of data that their operators process. Instead of relational tables (with set semantics), workflows process multi-dimensional arrays. An array has a schema to which each cell[1] conforms to. Array schemas distinguish between dimension and value attributes. The values of a cell's dimension attributes, termed a

[1]We denote a *cell* as the array equivalent of a relational record.

coordinate, uniquely identifies the cell; value attributes have no such restriction. For example, if the application stores satellite images of the earth, the dimension attributes may be latitude and longitude, and the value attributes may be the red, green and blue wavelength intensities.

Locality

Scientific applications typically process data that models physical world, and consequently have a natural notion of locality (e.g., latitude, longitude, time, voltage). These properties can help constrain the types of lineage relationships between workflow inputs and outputs so that we can develop efficient ways to represent the relationships.

3.2.2 WHY SCIENTIFIC DATA PROCESSING?

Throughput

The relative overhead of capturing fine-grained lineage fundamentally depends on the data-processing throughput of the workflow execution system. By this yardstick, scientific systems offer a particualrly challenging scenario given their high-throughput nature. As a simple example, consider a system that only processes 1000 records at 1 record/second. The lineage system can spend 1 minute to compute and store lineage metadata and incur a modest 6% runtime penalty. On the other hand, if the system throughput is 1000 records/second, then the same lineage overhead causes a 6,000% runtime slowdown!

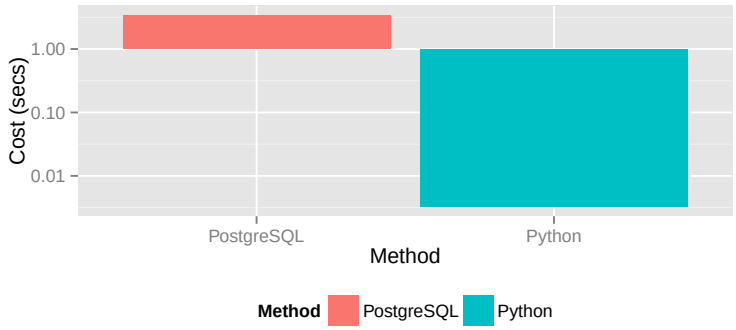


Figure 3-1: Cost of incrementing one million floats in PostgreSQL and Python+Numpy.

Figure 3-1 shows the results of a simple benchmark comparing PostgreSQL and Python+Numpy for incrementing one million float values by one. The dataset is stored in a single-column table as one million records in PostgreSQL and as a million cell NumPy array in Python.

There is a 3 orders-of-magnitude difference between the two approaches. Although not all of the difference can be attributed the difference between iterator-based and vector-based execution, it is clear that there is a large disparity in per-record processing times between the two types of systems.

Note that many scientific applications use highly optimized matrix libraries such as ScaLAPACK [31] that are significantly faster than Python+NumPy. The process costs in these applications will be even faster, and thus, ability to manage the resource costs is even more crucial.

User Defined Operators

Most workflow systems such as Hadoop [110], Spark [122], and scientific systems support custom operators in the form of user-defined functions. The lineage system depends on the developer to instrument the custom operator to export the internal lineage information to the lineage system through lineage API calls. However, the API design must be sufficiently efficient so that the amortized overhead is comparable to or less than the base operator execution costs. The microbenchmark in Figure 3-1 suggests that a low-overhead API designed for science applications will naturally be applicable in general record-based systems such as Hadoop or Spark.

Generality

The key concepts we used to design SubZero – physical independence, cost-based provenance materialization, and support for user defined functions – are applicable to workflow-based data processing systems irrespective of their data model or application domain. In fact, our system design is general enough to be easily extended for other non-scientific workflow-based systems. In addition, we present a simple but powerful lineage representation called *PayloadLineage* that can be used to implement many the lineage storage techniques in most existing fine-grained lineage systems. We further explore these relationships in the discussion (Section 3.10).

3.3 USE CASES

We developed two benchmark applications after discussions with environmental scientists, astronomists, and geneticists. The first is an image processing benchmark developed with scientists at the Large Synoptic Survey Telescope (LSST) project. It is very similar to environmental science requirements, so they are combined together. The second was developed

with geneticists at the Broad Institute[2]. Each benchmark consists of a workflow description, a dataset, and lineage queries. We used the benchmarks to design the optimizations described in this chapter. This section will briefly describe each benchmark's scientific application, the types of desired lineage queries, and application-specific insights.

3.3.1 ASTRONOMY

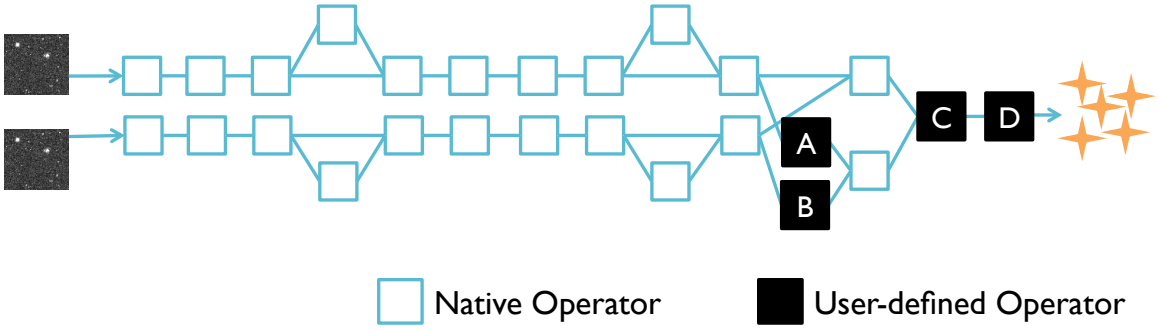


Figure 3-2: Diagram of LSST workflow. Each empty rectangle is a SciDB native operator while the black-filled rectangles A-D are UDFs.

The Large Synaptic Survey Telescope (LSST) is a wide angle telescope slated to begin operation in Fall 2015. A key challenge in processing telescope images is filtering out high energy particles (cosmic rays) that create abnormally bright pixels in the resulting image, which can be mistaken for stars. The telescope compensates by taking two consecutive pictures of the same piece of the sky and removing the cosmic rays in software. The LSST image processing workflow (Figure 3-2) takes two images as input and outputs an annotated image that labels each pixel with the celestial body it belongs to. It first cleans and detects cosmic rays in each image separately, then creates a single composite, cosmic-ray-free, image that is used to detect celestial bodies. There are 22 SciDB built-in operators (blue solid boxes) that perform common matrix operations, such as convolution, and four UDFs (red dotted boxes labeled A-D). The UDFs A and B output cosmic-ray masks for each of the images. After the images are subsequently merged, C removes cosmic-rays from the composite image, and D detects stars from the cleaned image.

The LSST scientists are interested in three types of queries. The first picks a star in the output image and traces the lineage back to the initial input image to detect bad input pixels. The latter two queries select a region of output (or input) pixels and trace the pixels backward (or forward) through a subset of the workflow to identify a single faulty

[2] http://www.broadinstitute.org/

operator. As an example, suppose the operator that computes the mean brightness of the image generated an anomalously high value due to a few bad pixel, which led to further mis-calculations. The astronomer might work backward from those calculations, identify the input pixels that contributed to them, and filter out those pixels that appear excessively bright.

Both the LSST and environmental scientists described workloads where the majority of the data processing code computes output pixels using input pixels within a small distance from the corresponding coordinate of the output pixel. These regions may be constant, pre-defined values, or easily computed from a small amount of additional metadata. For example, a pixel in the mask produced by cosmic ray detection (CRD) is set if the related input pixel is a cosmic ray, and depends on neighboring input pixels within a radius of 3 pixels. Otherwise, it only depends on the related input pixel. They also felt that it is sufficient for lineage queries to return a superset of the exact lineage. Although we do not take advantage of this insight, this suggests future work in lossy compression techniques.

3.3.2 GENOMICS PREDICTION

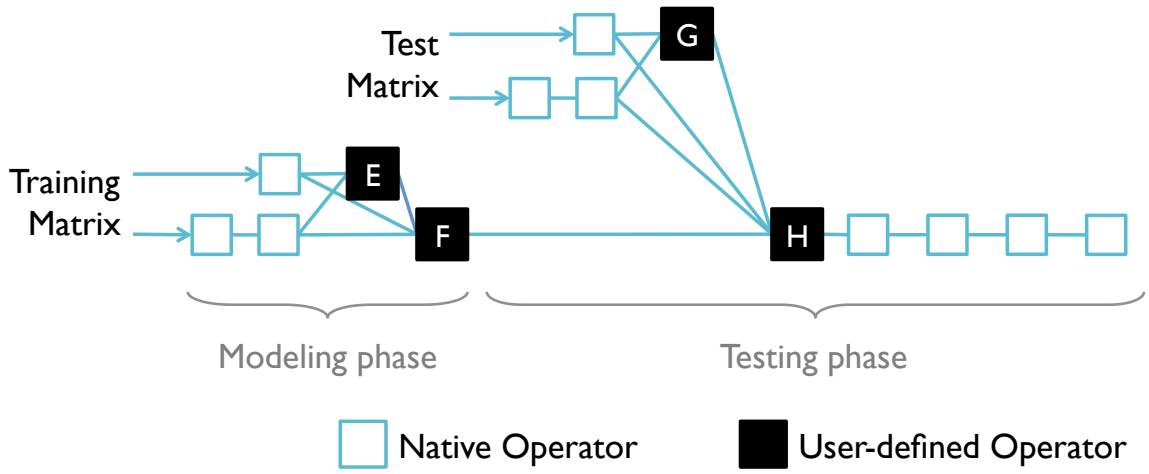


Figure 3-3: Simplified diagram of genomics workflow. Each empty rectangle is a SciDB native operator while the black filled rectangles are UDFs.

We have also been working with researchers at the Broad Institute on a genomics benchmark related to predicting recurrences of medulloblastoma in patients. Medulloblastoma is a form of cancer that spawns brain tumors that spread through the cerebrospinal fluid. Pablo et. al [105] have identified a set of patient features that help predict relapse in medulloblastoma patients that have been treated. The features include histology, gene expression levels, and the existence of genetic abnormalities. The workflow (Figure 3-3) is a

two-step process that first takes a training patient-feature matrix and outputs a Bayesian model. Then it uses the model to predict relapse in a test patient-feature matrix. The model computes how much each feature value contributes to the likelihood of patient relapse. The ten built-in operators (solid blue boxes) are simple matrix transformations. The remaining UDFs extract a subset of the input arrays (E,G), compute the model (F), and predict the relapse probability (H).

The model is designed to be used by clinicians through a visualization that generates lineage queries. The first query picks a relapse prediction and traces its lineage back to the training matrix to find supporting input data. The second query picks a feature from the model and traces it back to the training matrix to find the contributing input values. The third query points at a set of training values and traces them forward to the model, while the last query traces them to the end of the workflow to find the predictions they affected.

The genomics benchmark can devote up-front storage and runtime overhead to ensure fast query execution because it is an interactive visualization. Although this is application specific, it suggests that scientific applications have a wide range of storage and runtime overhead constraints.

3.4 ARCHITECTURE

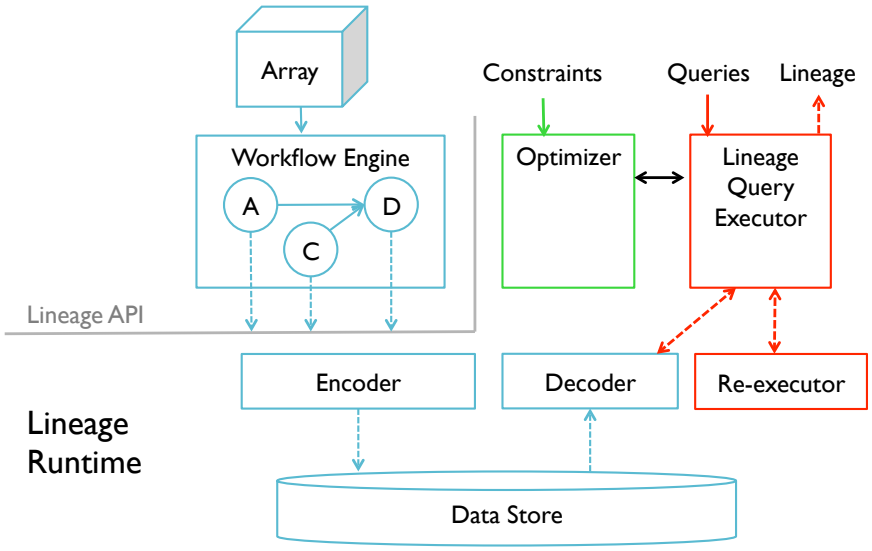


Figure 3-4: The SubZero architecture.

SubZero records and stores lineage data at workflow runtime and uses it to efficiently execute lineage queries. The input to SubZero is a workflow specification (the graph in *Workflow Engine*), constraints on the amount of storage that can be devoted to lineage

tracking and the amount of workflow slowdown the user is willing to tolerate, and a sample lineage query workload that the user expects to run. SubZero optimally decides the type of lineage that each operator in the workflow will generate (the *lineage strategy*) in order to maximize the performance of the expected query workload performance.

Figure 3-4 shows the system architecture. The solid and dashed arrows indicate the control and data flow, respectively. The solid gray line indicates the *Lineage API* that the *Workflow Engine* can call to access the SubZero lineage runtime. The colors distinguish components that are used while the system is capturing lineage data (blue), executing a lineage query (red), and running the SubZero optimizer (green).

Users interact with SubZero by defining and executing workflows (*Workflow Engine*), specifying storage and runtime constraints to the *Optimizer*, and running lineage queries (*Query Executor*). Each operator is additionally instrumented to list the region pair representations (described in Section 3.6) it can generate, which defines the set of optimization possibilities.

Each operator initially operates as a black-box (i.e., just records the names of the inputs it processes) but over time the optimizer will change the operator's strategy in terms of which operators should generate lineage and how it should be encoded and indexed. As operators process data, they use the *Lineage API* to write lineage data to the *Lineage Runtime*. The *Encoder* then serializes the lineage before writing it to *Operator Specific Datastores*. The *Runtime* may also send lineage and other statistics to the *Optimizer*, which calculates statistics such as the amount of lineage that each operator generates.

SubZero periodically runs the *Optimizer*, which uses an *Integer Programming Solver* to compute the new lineage strategy. On the right side, the *Query Executor* compiles lineage queries into query plans that join the query data with lineage data. The *Executor* requests lineage from the *Runtime*, which either reads and decodes materialized lineage or uses the *Re-executor* to re-run operators and generate non-materialized lineage. It also sends statistics (e.g., query fanout and fanin) to the optimizer that are used to refine future optimizations.

Given this overview, we now describe the different representations of fine-grained lineage that the system can record (Section 3.6), the functionality of the *Runtime*, *Encoder*, and *Query Executor* (Section 3.7), and finally the optimizer in Section 3.8.

3.5 LINEAGE REPRESENTATIONS

Section 2.4 presented the logical lineage data model. However, the naive representation of the logical model easily incurs very high resource overhead. To address this issue, this section describes three representations, including the lazy approach introduced in the introduction

that substantially reduce the overhead.

We have pre-instrumented SubZero all built-in matrix operators (e.g., addition, multiplication, convolution) to generate lineage information in all three representations, and provide an API for UDF designers to expose these relationships. If the API is not used, then SubZero assumes an all-to-all relationship between every cell in the input arrays and every cell in the output array.

3.5.1 CELL-LEVEL LINEAGE

Cell-level lineage is the naive approach that explicitly represents fine-grained lineage as a set of input and output cell pairs. Although we model and refer to lineage as a mapping between input and output cells, the SubZero implementation stores these mappings as references to physical cell coordinates.

3.5.2 BLACK-BOX LINEAGE

SubZero does not require additional resources to store black-box lineage because the workflow executor stores coarse-grained lineage by default. This is sufficient to re-run any previously executed operator from any point in the workflow. In this representation, the lineage is only materialized when the user executes a lineage query.

3.5.3 REGION LINEAGE

Scientific applications often exhibit locality where sets of output cells depend on the same set of input cells. For example, the LSST star detection operator finds clusters of adjacent bright pixels and generates an array that labels each pixel with the star that it belongs to. Every output pixel labeled *Star X* depends on all of the input pixels in the *Star X* region.

For this reason, it makes sense to explicitly represent this set-wise relationship using the region lineage representation. Region lineage represents fine-grained lineage a set of *region pairs*, where a region pair describes an all-to-all lineage relationship between a set of output cells $outcells$ and a set of input cells $incells_i$ in each input array, I_P^i:

$$\{(outcells, incells_1, ..., incells_n) | outcells \subseteq O_P \wedge incells_i \subseteq I_P^i\}$$

Region lineage is an improvement over cell-level lineage for two reasons. First, based on our experience instrumenting the two benchmark applications, region lineage is less cumbersome to express and keep track of than cell-level lineage, and results in less code to write. Second, region lineage is more resource efficient than cell-level lineage – in fact, region

lineage strictly outperforms cell-level lineage in all of the applications we have examined. For this reason, and to avoid redundant text, later sections will exclusively discuss region pairs.

3.6 LINEAGE API

SubZero helps developers write operators that efficiently represent and store lineage. Whereas the previous section introduced region lineage as part of the lineage data model, this section presents several concrete representations of region lineage and the APIs that UDF developers can use to generate lineage from within an operator. The next section will describe how the difference representations are encoded for physical storage.

This section also introduces the mechanism that the runtime uses to control the which lineage representation an operator should generate. Finally, we describe how SubZero re-executes black-box operators during a lineage query. Table 3-5 summarizes the runtime methods exposed to code within an operator. Table 3-6 summarizes operator methods that the developer overrides to add lineage support.

For ease of explanation, this section is described in the context of the LSST operator *CRD* (cosmic ray detection, depicted as A and B in Figure 3-2) that finds pixels containing cosmic rays in a single image, and outputs an array of the same size. If a pixel contains a cosmic ray, the corresponding cell in the output is set to 1, and the output cell depends on

Runtime API Method	Description
lwrite(outcells, incells_1, ...,incells_n)	API to store lineage relationship.
lwrite(outcells, payload)	API to store small binary payload instead of input cells. Called by payload operators.

Table 3-5: Runtime methods that SubZero makes available to the operators.

Operator API Method	Description
run(input_1,...,input_n,cur_reps)	Execute the operator, generating lineage types in cur_reps $\subseteq \{Full, Map, Pay, Comp, Blackbox\}$
map_b(outcell, i)	Computes the input cells in input_i that contribute to outcell.
map_f(incell, i)	Computes the output cells that depend on incell $\in \text{input}_i$.
map_p(outcell, payload, i)	Computes the input cells in input_i that contribute to outcell. This method has access to payload.
supported_representations()	Returns the representations $C \subseteq \{Full, Map, Pay$ $Comp, Blackbox\}$ that the operator can generate.

Table 3-6: Operator methods that the developer will override.

the 49 neighboring pixels within a 3 pixel radius. Otherwise the output cell is set to 0, and only depends on the corresponding input pixel. A region pair is denoted (*outcells*, *incells*).

3.6.1 BASIC OPERATOR STRUCTURE

The following code snippet is the basic structure of a SubZero operator:

```
class OpName:
  def run(input_1, ..., input_n, cur_reps):
    """
    Process the inputs, emit the output record
    lineage representations specified in cur_reps
    """
    pass

  def supported_representations():
    """
    Return the lineage representations the
    operator supports
    """
    pass
```

Each operator implements a *run()* method, which is called when inputs are available to be processed. It is passed a list of lineage representations it should output in the *cur_reps* argument; it writes out lineage data using the *lwrite()* method described below. The developer specifies the representations that the operator supports (and that the runtime will consider) by overriding the *supported_representations()* method. If the developer does not override *supported_representations()*, SubZero assumes an all-to-all relationship between the inputs and outputs. Otherwise, the operator automatically supports black-box lineage as well.

3.6.2 LINEAGE REPRESENTATIONS

SubZero supports four region lineage representations (*Full, Map, Pay, Comp*) and black-box lineage (*Blackbox*). *cur_reps* is set to *Blackbox* when the operator does not need to generate any pairs (because black box lineage is always in use). *Full* lineage explicitly stores all region pairs, and the other lineage representations reduce the amount of lineage that is stored by

partially computing lineage at query time using developer defined mapping functions. The following sections describe the representations in more detail.

Full Lineage

Full lineage (*Full*) explicitly represents and stores all region pairs. It is straightforward to instrument any operator to generate full lineage. The developer simply writes code that generates region pairs and uses *lwrite*() to store the pairs. For example, in the following CRD pseudocode, if *cur_reps* contains *Full*, the code iterates through each cell in the output, calculates the lineage, and calls *lwrite*() with lists of cell coordinates. Note that if *Full* is not specified, the operator can avoid running the lineage related code.

```
def run(image, cur_reps):
  if "Full" in cur_reps:
    for each cell in output:
      if cell == 1:
        neighs = get_neighbor_coords(cell)
        lwrite([cell.coord], neighs)
  else:
    lwrite([cell.coord], [cell.coord])
```

Although this lineage mode accurately records the lineage data, it is potentially very expensive to both generate and store. We have identified several widely applicable operator properties that allow the operators to generate more efficient representations of lineage, which we describe next.

Mapping Lineage

Mapping lineage (*Map*) compactly represents an operator's lineage using a pair of mapping functions. Many operators such as matrix transpose exhibit a fixed execution structure that does not depend on the input cell values. These operators, called *mapping operators*, can compute forward and backward lineage from a cell's coordinates and array metadata (e.g., input and output array sizes) and do not need to access array data values.

This is a valuable property because mapping operators do not incur runtime and storage overhead. For example, one-to-one operators, such as matrix addition, are mapping operators because an output cell only depends on the input cell at the same coordinate, regardless of the value. Developers implement a pair of mapping functions, $map_f(cell, i)/map_b(cell, i)$, that calculate the forward/backward lineage of an input/output cell's coordinates, with

respect to the i'th input array. For example, a 2D transpose operator would implement the following functions:

```
def map_b((x,y), i):
  return [(y,x)]

def map_f((x,y), i):
  return [(y,x)]
```

Most scientific operators (e.g., matrix multiply, join, transpose, convolution) are mapping operators, and we have implemented their forward and backward mapping functions. Mapping operators are depicted as the blue boxes in the astronomy (Figure 3-2) and genomics (Figure 3-3) workflows.

Payload Lineage

Rather than storing the input cells in each region pair, payload lineage (Pay) stores a small amount of data (*a payload*), and recomputes the lineage using a payload-aware mapping function ($map_p()$). Unlike mapping lineage, the mapping function has access to the user-stored binary payload. This mode is particularly useful when the operator has high fanin and the payload is very small.

For example, suppose that the radius of neighboring pixels that a cosmic ray pixel depends on increases with brightness, then payload lineage only stores the brightness insteall of the input cell coordinates. (*Payload operators*) call $lwrite(outcells, payload)$ to pass in a list of output cell coordinates and a binary blob, and define a *payload function*, $map_p(outcell, payload, i)$, that directly computes the backward lineage of $outcell \in outcells$ from the *outcell* coordinate and the payload. The result are input cells in the i'th input array. As with mapping functions, payload lineage does not need to access array data values. The following pseudocode stores radius values instead of input cells:

```
def run(image, cur_reps):
  if "Pay" in cur_reps:
    for each cell in output:
      if cell == 1:
        lwrite([cell.coord], '3')
      else:
        lwrite([cell.coord], '0')
```

```
def map_p((x,y), payload, i):
  return get_neighbors((x,y), int(payload))
```

In the above implementation, each region pair stores the output cells and an additional argument that represents the radius, as opposed to the neighboring input cells. When a backward lineage query is executed, SubZero retrieves the (outcells, payload) pairs that intersect with the query and executes map_p on each pair. This approach is particularly powerful because the payload can store arbitrary data – anything from array data values to lineage predicates [56].i Thus, existing lineage systems such as that in Trio [113] and Panda [56] can be readily implemented is SubZero. Operators D to G in the two benchmarks (Figures 3-2 and 3-3) are payload operators.

Note that payload functions are designed to optimize execution of backward lineage queries. While SubZero can index the input cells in full lineage, the payload is a binary blob that cannot be easily indexed. A forward query must iterate through each (outcells, payload) pair and compute the input cells using map_p before it can be compared to the query coordinates.

Composite Lineage

Composite lineage (*Comp*) composes mapping and payload lineage. The mapping function defines the default relationship between input and output cells, and results of the payload function *overwrite* the default lineage if specified. For example, CRD can represent the default relationship – each output cell depends on the corresponding input cell in the same coordinate – using a mapping function, and write payload lineage for the cosmic ray pixels:

```
def run(image,cur_reps):
  if "Comp" in cur_reps:
    for each cell in output:
      if cell == 1:
        lwrite([cell.coord], 3)
      else:
        # map_b defines default behavior
        pass

def map_p((x,y), radius, i):
  return get_neighbors((x,y), radius)
```

```
def map_b((x,y), i):
  return [(x,y)]
```

Composite operators can avoid storing lineage for a significant fraction of the output cells. Although it is similar to payload lineage in that the payload cannot be indexed to optimize forward queries, the amount of payload lineage that is stored may be small enough that iterating through the small number of (outcells, payload) pairs is efficient. Operators A,B and C in the astronomy benchmark (Figure 3-2) are composite operators.

Note that a more general layered approach is possible, where the user defines n layers of lineage representations and a higher layer overwrites the lineage represented by a lower layer. In such a model, our composite lineage is a special case where $n = 2$. From our experience, we have not encountered operators that warrant the added complexity.

3.6.3 OPERATOR RE-EXECUTION

An operator stores black-box lineage when *cur_reps* equals *Blackbox*. When SubZero executes a lineage query on an operator that stored black-box lineage, the operator is re-executed in tracing mode. When the operator is re-run at lineage query time, SubZero passes $cur\_reps = Full$, which causes the operator to perform $lwrite()$ calls. The arguments to these calls are sent to the lineage query executor.

In order for re-execution to be correct (the lineage is identical to capturing the lineage when the operator was first executed), operators need to be deterministic. In our execution setting, determinism can be enforced by instrumenting every non-deterministic Python runtime call and replay their results during re-execution.

Selective Re-execution

Rather than re-executing the operator on the full input arrays, SubZero could also reduce the size of the inputs by applying bounding box predicates prior to re-execution. The predicates would reduce both the amount of lineage that needs to be stored and the amount of data that the operator needs to re-process.

We considered this approach and extended both mapping and full operators to compute and store bounding box predicates. Unfortunately, we did not find it to be a widely useful optimization. During query execution, SubZero must retrieve the bounding boxes for every query cell, and either re-execute the operator over each cell's corresponding bounding box, or merge the bounding boxes for every cell and re-run the operator using the merged bounding box predicate. Unfortunately, the former approach incurs an overhead on each execution (to read the input arrays and apply the predicates) that quickly becomes a significant cost. In

the latter approach, the merged bounding box quickly expands to encompass the full input array, which is equivalent to completely re-executing the operator, but incurs the additional cost to retrieve the predicates. For these reasons, we did not further consider this approach.

3.7 IMPLEMENTATION

This section describes the *Runtime*, *Encoder*, and *Query Executor* components in greater detail.

3.7.1 RUNTIME

In SciDB (and our prototype), we automatically store black-box lineage by using write-ahead logging, which guarantees that black-box lineage is written before the array data, and is "no overwrite" on updates. Region lineage is stored in a collection of BerkeleyDB hashtable instances. We use BerkeleyDB to store region lineage to avoid the client-server communication overhead of interacting with traditional DBMSes. We turn off fsync, logging and concurrency control to avoid recovery and locking overhead. This is safe because the region lineage is treated as a cache, and can always be recovered by re-running operators.

The runtime allocates a new BerkeleyDB database for each operator instance that stores region lineage. Blocks of region pairs are buffered in memory, and bulk encoded using the *Encoder*. The data in each region pair is stored as a unit (SubZero does not optimize across region pairs), and the output and input cells use separate encoding schemes. The layout can be optimized for backward (forward) queries by storing the output (input) cells as the hash key. On a key collision, the runtime decodes, merges, and re-encodes the two hash values. The next subsection describes how the *Encoder* serializes the region pairs.

3.7.2 ENCODER

While Section 3.6 presented efficient ways to represent region lineage, SubZero still needs to store cell coordinates, which can easily be larger than the original data arrays. The *Encoder* stores the input and output cells of a region pair (generated by calls to $lwrite()$) into one or more hash table entries, specified by an *encoding strategy*. We say the encoding strategy is *backward optimized* if the output cells are stored in the hash key, and *forward optimized* if the hash key contains input cells.

We found that four basic strategies work well for the operators we encountered. – $FullOne$ and $FullMany$ are the two strategies to encode full lineage, and $PayOne$ and $PayMany$ encode payload lineage.

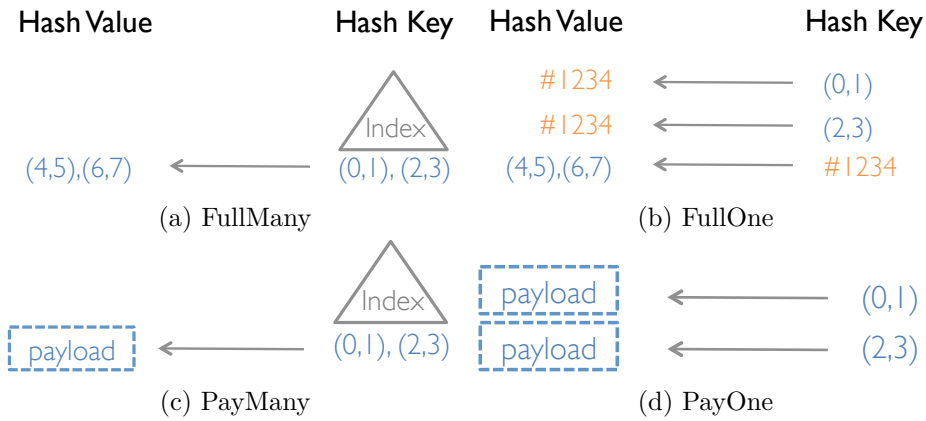


Figure 3-7: Four examples of encoding strategies

Figure 3-7 depicts how the backward-optimized implementation of these strategies encode a single region pair consisting of two output cells with coordinates $(0, 1)$ and $(2, 3)$ that depend on two input cells with coordinates $(4, 5)$ and $(6, 7)$.

FullMany

FullMany uses a single hash entry with the set of serialized output cells as the key and the set of input cells as the value (Figure 3-7a). Each coordinate is bitpacked into a single integer if the array is small enough. We also create an R*-tree on the cells in the hash key to quickly find the entries that intersect with the query. This index uses the dimensions of the array as its keys and identifies the hash table entries that contain cells in particular regions. The figure shows the unserialized versions of the cells for simplicity. *FullMany* is most appropriate when the lineage has high fanout because it only needs to store the output cells once.

FullOne

If the fanout is low, *FullOne* more efficiently serializes and stores each output cell as the hash key of a separate hash entry. The hash value stores a reference to a single entry containing the input cells (Figure 3-7b). This implementation doesn't need to compute and store bounding box information and doesn't need the spatial index because each input cell is stored separately, so queries execute using direct hash lookups.

PayMany and PayOne

For payload lineage, $PayMany$ stores the lineage in a similar manner as $FullMany$, but stores the payload as the hash value (Figure 3-7c). $PayOne$ creates a hash entry for every output cell and stores a duplicate of the payload in each hash value (Figure 3-7d).

Alternative Approaches

We tried a number of possible serialization techniques and found that complex encodings incur inordinately high encoding costs without noticeably reduced storage costs. Thus we don't present these techniques in the experimental results. Some of the techniques include:

1. Compute and store the bounding box of a set of cells, C, along with cells in the bounding box but not in C.
2. Logically partition the $N \times M$ array into a coarse $\frac{N}{gridsize} \times \frac{M}{gridsize}$ grid. For each grid cell that contains a cell in the lineage, record that the grid cell is *active* as well as the corresponding offsets within the grid cell that are part of the lineage. If the entire grid cell is part of the lineage, set a special bit instead of explicitly storing every offset.
3. Run-length encode the cells in row-major or column-major order.
4. *gzip* compress the resulting BerkeleyDB file. This method is effective at compressing the database file by up to 3× (on a synthetic, highly structured data file). However the resulting file cannot be directly queried and must be decompressed first.

3.7.3 LINEAGE AND STORAGE STRATEGY

The *Optimizer* picks a *Lineage Strategy* that spans the entire workflow instance. It picks one or more *Storage Strategies* for each operator. Each storage strategy is fully specified by the tuple:

$$(Representation, Encoding, Direction)$$

Where:

$$Representation \in \{Full, Map, Pay, Comp, Blackbox\} \tag{3.1}$$

$$Encoding \in \{FullMany, FullOne, PayMany, PayOne\} \tag{3.2}$$

$$Direction \in \{\leftarrow, \rightarrow\} \tag{3.3}$$

For example, $(Payload, PayMany, \leftarrow)$ will generate payload lineage, encode it using $PayMany$, and optimize the storage for backward lineage queries. SubZero can use multiple storage strategies for each operator to optimize for different query types.

3.7.4 QUERY EXECUTION

The *Query Executor* iteratively executes each step in the lineage query path by joining the lineage with the coordinates of the query cells, or the intermediate cells generated from the previous step. The output at each step is a set of cell coordinates that is compactly stored in an in-memory boolean array with the same dimensions as the input (backward query) or output (forward query) array. A bit is set if the intermediate result contains the corresponding cell. For example, suppose we have an operator P that takes as input a 1×4 array. Consider a backwards query asking for the lineage of some output cell C of P. If the result of the query is 1001, this means that C depends on the first and fourth cell in P's input.

We chose the in-memory array because many operators have large fanin or fanout, and can easily generate several times more results (due to duplicates) than are unique. De-duplication avoids wasting storage and saves work. Similarly, the executor can close an operator early if it detects that all of the possible cells have been generated.

Entire Array Optimization

We also implement an *entire array optimization* to speed up queries where all of the bits in the boolean array are set. For example, this can happen if a backward query traverses several high-fanin operators or an all-to-all operator such as matrix inversion. In these cases, calculating the lineage of every query cell is very expensive and often unnecessary. Many operators (e.g., matrix multiply or inverse) can safely assume that the forward (backward) lineage of an entire input (output) array is the entire output (input) array. This optimization is valuable when it can be applied – it improved the query performance of a forward query in the astronomy benchmark that traverses an all-to-all-operator by 83×.

In general, it is difficult to automatically identify when the optimization's assumptions hold. Consider a concatenate operator that takes two 2D arrays A, B with shapes (1, n) and (1, m), and produces an (1, n+m) output by concatenating B to A. The optimization would produce different results, because A's forward lineage is only a subset of the output. We currently rely on the programmer to manually annotate operators where the optimization can be applied.

3.8 LINEAGE STRATEGY OPTIMIZER

Having described the basic storage strategies implemented in SubZero, we now describe our lineage storage optimizer. The optimizer's objective is to choose a set of *storage strategies* that minimize the cost of executing the workflow while keeping storage overhead within user-defined constraints. We formulate the task as an integer programming problem, where the inputs are a list of operators, strategy pairs, disk overheads, query cost estimates, and a sample workload that is used to derive the frequency with which each operator is invoked in the lineage workload. Additionally, users can manually specify operator specific strategies prior to running the optimizer.

The formal problem description is stated as:

$$
\begin{array}{lll}
\min_x & \sum_i p_i * \left(\min_{j|x_{ij}=1} q_{ij}\right) + & \epsilon * \sum_{ij}(disk_{ij} + \beta * runtime_{ij}) * x_{ij} \\
\text{s.t.} & \sum_{ij} disk_{ij} * x_{ij} & \leq disk_{max} \\
 & \sum_{ij} runtime_{ij} * x_{ij} & \leq runtime_{max} \\
 & \forall_i \left(\sum_{0 \leq j < M} x_{ij}\right) & \geq 1 \\
 & \forall_{i,j} x_{ij} & \in \{0, 1\} \\
 & & \\
 & \text{user specified strategies} & \\
 & x_{ij} = 1 & \forall_{i,j} x_{ij} \in U
\end{array}
$$

Here, $x_{ij} = 1$ if operator i stores lineage using strategy j, and 0 otherwise. $disk_{max}$ is the maximum storage overhead specified by the user; q_{ij}, $runtime_{ij}$, and $disk_{ij}$, are the average query cost, runtime overhead, and storage overhead costs for operator i using strategy j as computed by the cost model. p_{ij} is the probability that a lineage query in the workload accesses operator i, and is computed from the sample workload. A single operator may store its lineage data using multiple strategies.

The goal of the objective function is to minimize the cost of executing the lineage workload, preferring strategies that use less storage. When an operator uses multiple strategies to store its lineage, the query processor picks the strategy that minimizes the query cost. The min statement in the left hand term picks the best query performance from the strategies that have been picked ($j|x_{ij} = 1$). The right hand term penalizes strategies that take excessive disk space or cause runtime slowdown. β weights runtime against disk overhead, and ϵ is set to a very small value to break ties. A large ϵ is similar to reducing $disk_{max}$ or $runtime_{max}$.

We heuristically remove configurations that are clearly non-optimal, such as strategies that exceed user constraints, or are not properly indexed for any of the queries in the

Strategy	Description
Astronomy Benchmark	
BlackBox	All operators store black-box lineage
BlackBoxOpt	Like BlackBox, uses mapping lineage for built-in-operators.
FullOne	Like BlackBoxOpt, but uses FullOne for UDFs.
FullMany	Like FullOne, but uses FullMany for UDFs.
Subzero	Like FullOne, but stores composite lineage using PayOne for UDFs.
Genomics Benchmark	
BlackBox	UDFs store black-box lineage
FullOne	UDFs store backward optimized FullOne
FullMany	UDFs store backward optimized FullMany
FullForw	UDFs store forward optimized FullOne
FullBoth	UDFs store FullForw and FullOne
PayOne	UDFs store PayOne
PayMany	UDFs store PayMany
PayBoth	UDFs store PayOne and FullForw

Table 3-8: Lineage Strategies for Benchmark Experiments.

workload (e.g., forward optimized when the workload only contains backward queries). The optimizer also picks mapping functions over all other classes of lineage.

We solve the ILP problem using the simplex method in GNU Linear Programming Kit. The solver's performance characteristics have been well studied [1] and takes about 1ms to solve for our science benchmarks.

3.8.1 QUERY-TIME OPTIMIZER

While the lineage strategy optimizer picks the optimal lineage strategy, the executor must still pick between accessing the lineage stored by one of the lineage strategies, or re-running the operator. The query-time optimizer consults the cost model using statistics gathered during query execution and the size of the query result so far to pick the best execution method. In addition, the optimizer monitors the time to access the materialized lineage. If it exceeds the cost of re-executing the operator, SubZero dynamically switches to re-running the operator. This bounds the worst case performance to 2× the black-box approach.

3.9 EXPERIMENTS

In the following subsections, we first describe how SubZero optimizes the storage strategies for the real-world benchmarks described in Section 3.3, then compare several of our lineage storage techniques with black-box level only techniques. The astronomy benchmark shows how our region lineage techniques improve over cell-level and black-box strategies on a sparse image processing workflow. The genomics benchmark illustrates the complexity in determining an optimal lineage strategy and the value of using an optimizer.

The SubZero prototype is written in Python and uses BerkeleyDB for the persistent store, and libspatialindex for the spatial index. We don't believe the choice of language is a affects our main conclusions because the main bottlenecks are storage, rather than CPU, related. The microbenchmarks are run on a 2.3 GHz linux server with 24 GB of RAM, running Ubuntu 2.6.38-13-server. The benchmarks are run on a 2.3 GHz MacBook Pro with 8 GB of RAM, a 5400 RPM hard disk, running OS X 10.7.2.

Overall, our findings are that:

- An optimal strategy heavily relies on operator properties such as fanin, and fanout, the specific lineage queries, and query execution-time optimizations. The difference between a sub-optimal and optimal strategy can be so large that an optimizer-based approach is crucial.

- Payload, composite, and mapping lineage are extremely effective and low overhead techniques that greatly improve query performance, and are applicable across a number of scientific domains. In particular, the composite technique can exploit applications with sparse arrays (such as astronomy datasets) to reduce the amount of payload lineage to store.

- SubZero can improve the LSST benchmark queries by up to 10× compared to naively storing the region lineage (similar to what cell-level approaches would do) and up to 255× faster than black-box lineage. The runtime and storage overhead of the optimal scheme is up to 30 and 70× lower than cell-level lineage, respectively, and only 1.49 and 1.95× higher than executing the workflow.

- Even though the genomics benchmark executes operators very quickly, SubZero can find the optimal mix of black-box and region lineage that scales to the amount of available storage. SubZero uses a black-box only strategy when the available storage is small, and switches from space-efficient to query-optimized encodings with looser constraints. When the storage constraints are unbounded, SubZero improves forward queries by over 500× and backward queries by 2-3×.

3.9.1 ASTRONOMY BENCHMARK

In this experiment, we run the Astronomy workflow with five backward queries and one forward query as described in Section 3.3.1. The 22 built-in operators are all expressed as mapping operators and the UDFs consist of one payload operator that detects celestial bodies and three composite operators that detect and remove cosmic rays. This workflow exhibits considerable locality (stars only depend on neighboring pixels), sparsity (stars are rare and small), and the queries are primarily backward queries. Each workflow execution consumes two 512×2000 pixel (8MB) images (provided by LSST) as input, and we compare the strategies in Table 3-8.

Overhead

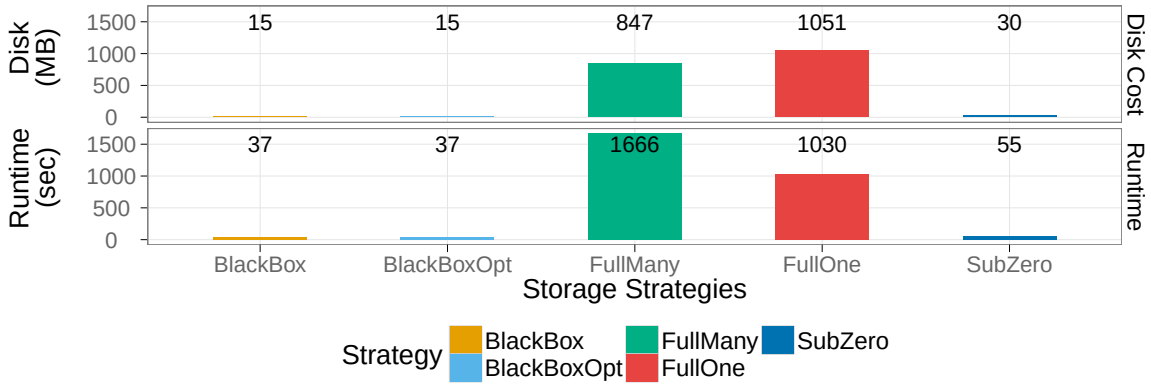


Figure 3-9: Astronomy Benchmark: disk and runtime overhead.

Figure 3-9 plots the disk and runtime overhead for each of the strategies. *BlackBox* and *BlackBoxOpt* identically show the base cost to execute the workflow and the size of the input arrays – the goal is to be as close to these bars as possible.

FullOne and *FullMany* both require considerable storage space (66×, 53×) because the three cosmic ray operators generate a region pair for every input and output pixel at the same coordinates. The runtime overhead is closely related to the disk costs, both *Full* approaches impact the workflow execution the most (6× and 44×, respectively.) Despite using less storage space *FullMany* has a higher runtime overhead to account for constructing the spatial index on the output cells.

The SubZero optimizer instead picks composite lineage that only stores payload lineage for the small number of cosmic rays and stars. This reduces the runtime and disk overheads to 1.49× and 1.95× the workflow inputs. By comparison, the intermediate and final result

arrays amount to 11.5× the workflow inputs, and thus the lineage storage overhead is comparably negligible.

Query Performance

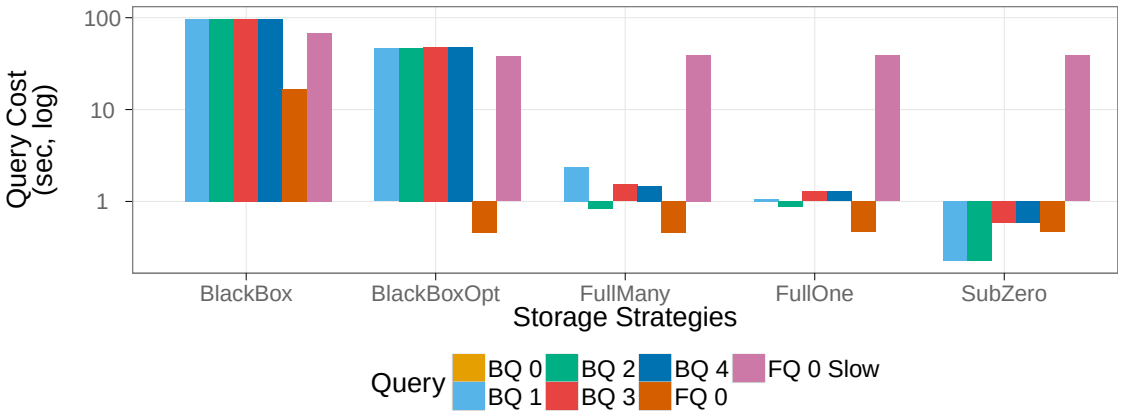


Figure 3-10: Astronomy Benchmark: query costs.

Figure 3-10 compares lineage query execution costs. *BQ x* and *FQ x* respectively stand for backward and forward query x. *FQ0Slow* executes the lineage query as normal, whereas the rest of the queries use the entire array optimization described in Section 3.7.4. Comparing *FQ0Slow* and *FQ0*, the all-to-all optimization improves the query performance by 83× because it can completely avoid the overhead of fine-grained lineage once every array cell is part of the query. A natural extension is to statically determine if a lineage query includes an intermediate all-to-all operator along its path, and switch to coarse-grained lineage if it is safe (Section 3.7.4).

BlackBox must re-run each operator and takes up to 100 secs per query. The difference bteween *BlackBox* and its runtime in Figure 3-9 constitutes the overhead of capturing lineage from every operator. *BlackBoxOpt* can avoid rerunning the mapping operators, but still re-runs and captures lineage from the computationally intensive UDFs.

Storing region lineage reduces the cost of executing the backward queries by 34× (*FullMany*) and 45× (*FullOne*) on average. SubZero benefits further by only reading lineage data for the array cells that contain stars or cosmic rays, and executing mapping functions for the majority of the cells. This allows it to execute 255× faster on average.

3.9.2 GENOMICS BENCHMARK

In this experiment, we run the genomics workflow and execute a lineage workload with an equal mix of forward and backward lineage queries (Section 3.3.2). There are 10 built-in mapping operators, and the 4 UDFs are all payload operators. In contrast to the astronomy workflow, these UDFs do not exhibit significant locality, and perform data shuffling and extraction operations that are not amenable to mapping functions. In addition, the operators perform fast and simple calculations so there is a less pronounced trade off between re-executing the workflow and accessing region lineage. In fact, there are cases where using the materialized lineage data is *slower* than the black box approach.

The dataset provided to us is a 56×100 matrix of 96 patients and 55 health and genetic features. Although the dataset is small, its structure is representative of similar datasets such as microarray gene expression data. Additionally, future datasets are expected to come from a larger group of patients, so we constructed larger datasets by replicating the patient data. The query performance and overheads scaled linearly with the size of the dataset (since costs primarily scale with respect to the size of the lineage) and so we report results for the dataset scaled by 100×.

The goal of this experiment is to explore the value of using a query optimizer as compared to picking a single static storage strategy for all of the operators. We find that the best storage strategy depends on a large number of factors including the operator runtime, lineage fanin and fanout, encoding costs, and user constraints.

We first compare several different static strategies (Table 3-8) with and without the query-time optimizer (Section 3.8.1) and then show how varying user constraints changes how the optimizer picks lineage strategies.

Query-Time Optimizer

This experiment compares the strategies in Table 3-8 with and without the query-time optimization described in Section 3.8.1. Each operator uses mapping lineage if possible, and otherwise stores lineage using the specified strategy. The majority of the UDFs generate region pairs that contain a single output cell. As mentioned in previous experiments, payload lineage stores very little binary data, and incurs less overhead than the full lineage approaches (Figure 3-11). Storing both forward and backward-optimized lineage ($PayBoth$ and $FullBoth$) requires significantly more overhead – 8 and 18.5× more space than the input arrays, and a corresponding 2.8 and 26× runtime slowdown.

Figure 3-12a highlights how query performance can *degrade* if the executor blindly joins queries with mismatched indexed lineage (e.g., backward-optimized lineage with forward

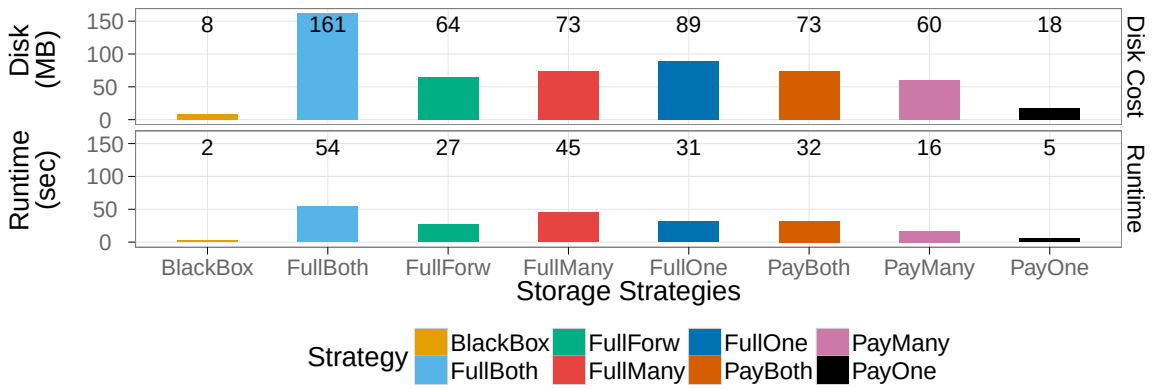


Figure 3-11: Genomics benchmark: disk and runtime overhead.

queries)[3]. For example, *FullForw* degraded backward query performance by up to 520×. For example, BQ1 ran slower because the query path contains several large fanin operators, which generates so many intermediate results that performing index lookups on each intermediate result is slower than re-running the operators. Finally, the forward optimized strategies improved the performance of $FQ0$ and $FQ2$ because the fanout is low.

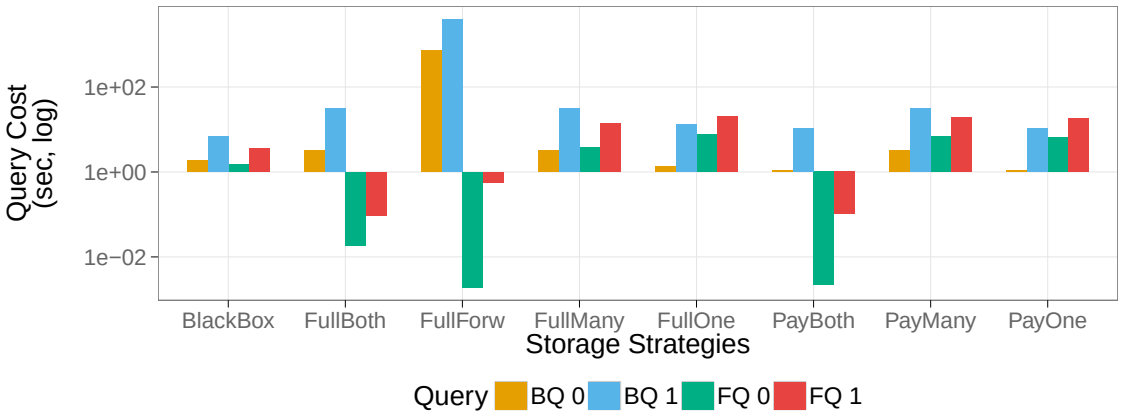
Figure 3-12b – note the different domain of the Y-axis – shows that the query-time optimizer executes the queries as fast as, or faster than, *BlackBox*. In general this cannot be guaranteed because it requires accurate statistics and cost estimation [77], however the optimizer can limits the query performance degradation to 2× by dynamically switching to the *BlackBox* strategy. Overall, the backward and forward queries improved by up to 2 and 25×, respectively.

Lineage Strategy Optimizer

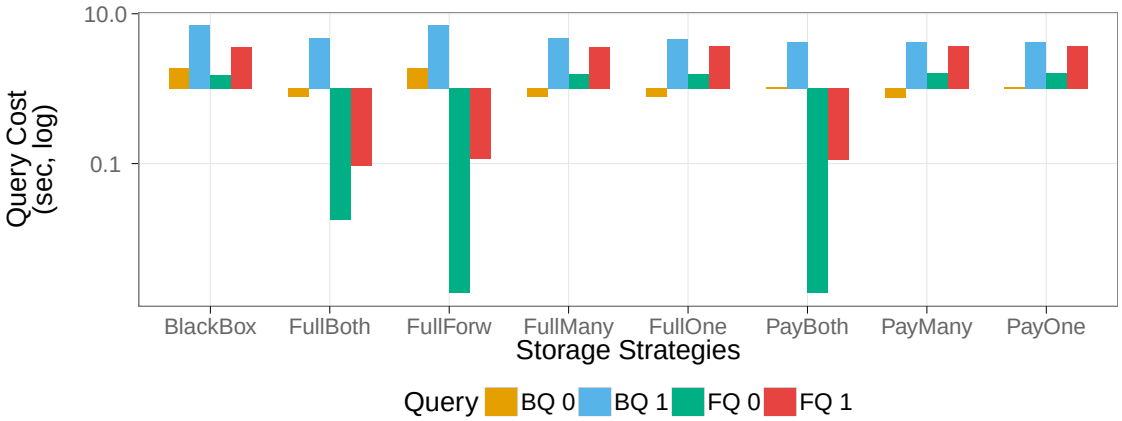
The above experiments compared many static strategies, each with different performance characteristics depending on the operator and query, and found that picking storage strategies on a per-operator basis is valuable. We now evaluate the SubZero optimizer on the genomics benchmark by ignoring the runtime constraint and varying the storage constraint from 1MB (only stores the input arrays) to 100MB (effectively unconstrained).

In these experiments we do not set a bound on the runtime overhead for two reasons. First, as we will see, the runtime overhead correlates with the storage costs so the graphs would be very similar (albeit scaled). Second, applications are typically willing to tolerate 50% to 200% runtime slowdown, however given those constraints SubZero would consistently

[3]All comparisons are relative to *BlackBox*



(a) Without query-time optimizer (Y-axis ranges from 1e-02 to 1000.)



(b) With query-time optimizer (Y-axis ranges from 1e-03 to 10.)

Figure 3-12: Genomics benchmark: query costs with and without the query-time optimizer (Section 3.8.1.)

choose the *BlackBox* strategy, which does not reveal any insights.

Figures 3-13 and 3-14 illustrate that SubZero can successfully pick more storage intensive strategies that are predicted to improve the benchmark queries as the storage constraint is relaxed. SubZero chooses *BlackBox* when the constraint is too small (<20MB), and stores forward and backward-optimized lineage that benefits all of the queries when the minimum amount of storage is available (20MB). Materializing further lineage has diminishing storage-to-query benefits. With 100MB, SubZero uses 50MB to forward-optimize the UDFs using $(MANY, ONE)$, which reduces the forward queries to sub-second latencies. This is because the UDFs have low fanout, so each join in the query path is a small number of hash lookups.

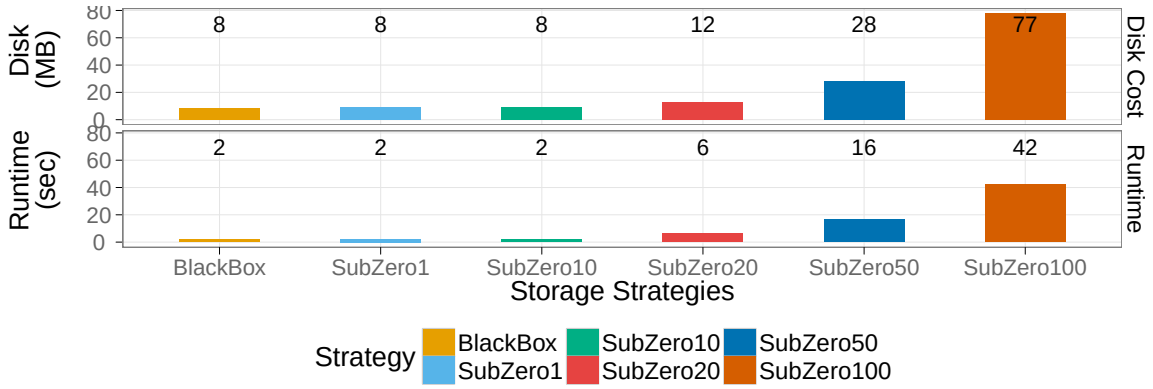


Figure 3-13: Genomics benchmark: disk and runtime overhead when varying SubZero storage constraints.

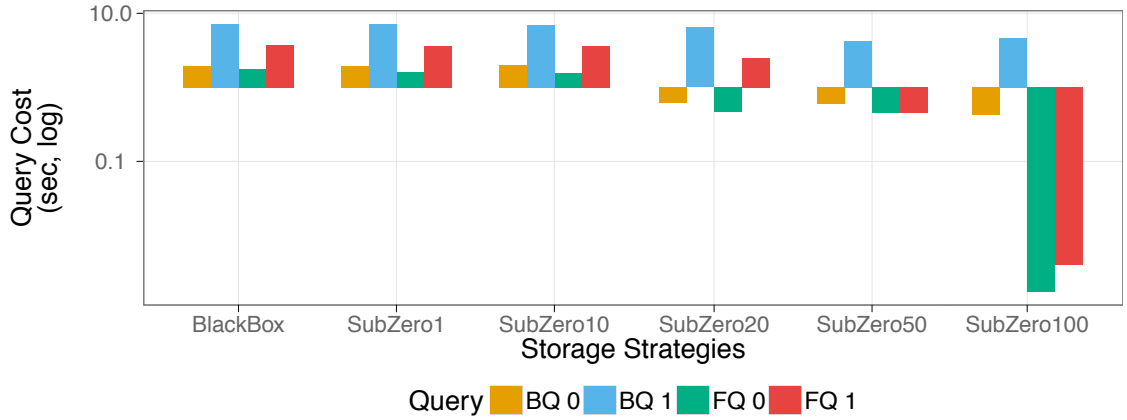


Figure 3-14: Genomics benchmark: query costs when varying SubZero storage constraints.

3.9.3 MICROBENCHMARKS

It it can be difficult to distinguish the sources of benefits in the above end-to-end benchmark experiments. The following experiments explore the key differences between the prevailing strategies in terms of overhead and query performance. The comparisons use an operator that generates synthetic lineage data with tunable parameters. We will show results from varying the dominant parameters – fanin, fanout and payload size (for payload lineage).

Experiment Setup

Each experiment processes and outputs a 3.8MB 1000x1000 array, and generates lineage for 10% of the output cells. The results scaled close to linearly as the number of output

cells with lineage varies. A region pair is randomly generated by selecting a cluster of output cells with a radius defined by $fanout$, and selecting $fanin$ cells in the same area from the input array. We generate region pairs until the total number of output cells is equal to 10% of the output array. The payload strategy uses a payload size of $4 \times fanin$ bytes (the payload is expected to be very small). We compare several backward optimized strategies ($\leftarrow FullMany$, $\leftarrow FullOne$, $\leftarrow PayMany$, $\leftarrow PayOne$), one forward lineage strategy ($\rightarrow FullOne$), and black-box ($BlackBox$). We first discuss the overhead to store and index the lineage, then comment on the query costs.

Overhead

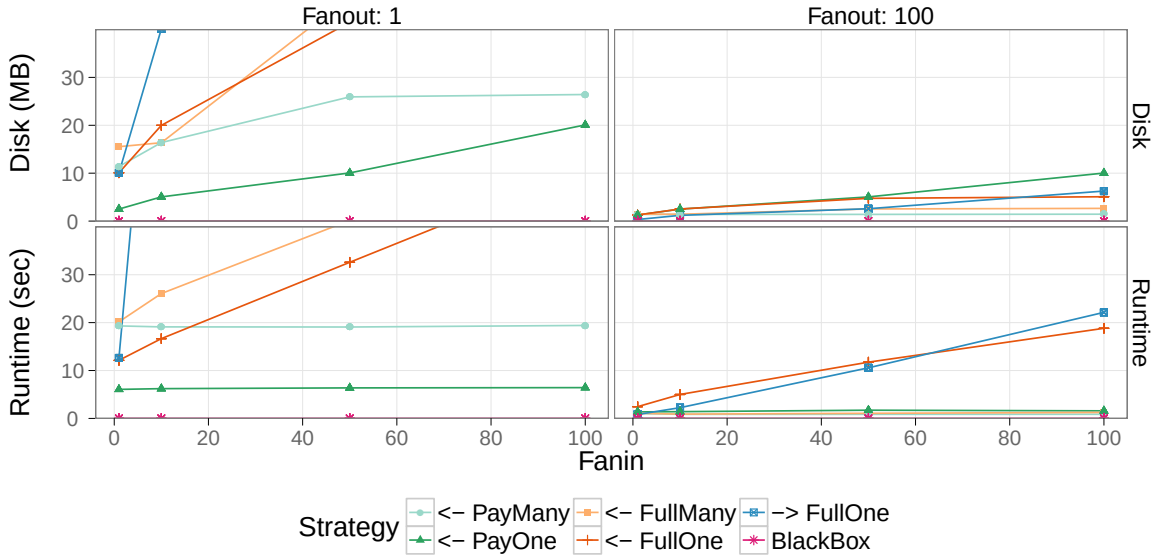


Figure 3-15: Microbenchmarks: disk and runtime overhead

Figure 3-15 compares the runtime and disk overhead of the different strategies. The best full lineage strategy differs based on the operator fanout. $FullOne$ is superior when $fanout \leq 5$ because it doesn't need to create and store the spatial index. The crossover point to $FullMany$ occurs when the cost of duplicating hash entries for each output cell in a region pair exceeds that of the spatial index. The overhead of both approaches increases with fanin. In contrast, payload lineage has a much lower overhead than the full lineage approaches and is independent of the fanin because the payload is typically small and does not need to be encoded. When the fanout increases to 50 or 100, $PayMany$ and $FullMany$ require less than 3MB and 1 second of overhead. The forward optimized $FullOne$ is comparable to the other approaches when the fanin is low. However, when the fanin increases it can require up

to $fanin\times$ more hash entries because it creates an entry for every distinct input cell in the lineage. It converges to the backward optimized $FullOne$ when the fanout and fanin are high. Finally, $BlackBox$ has nearly no overhead.

Query Performance

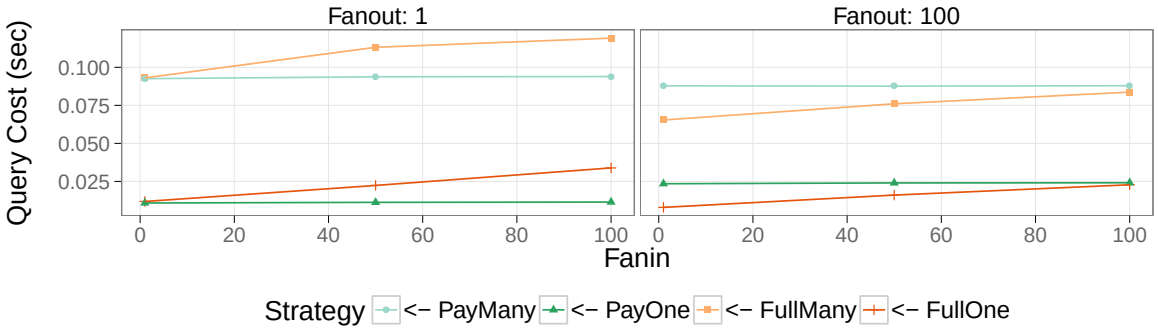


Figure 3-16: Microbenchmarks: backward lineage queries, only backward-optimized strategies

This experiment (Figure 3-16) shows the costs of executing a backward lineage query when the storage strategy is backward-optimized and the operator fanin and fanout are varied. The query performance scales almost linearly with the number of cells so we fix the number of cells at 1000.

There is a clear difference between $FullMany$ or $PayMany$, and $FullOne$ or $PayOne$, due to the additional cost of accessing the spatial index. Payload lineage performs independently of the fanin, and is similar to, but not consistently faster than, Full lineage. Finally (not shown), using a mis-matched index (e.g, using forward-optimized lineage for backward queries) slows query performance by up to two orders of magnitude as compared to $BlackBox$.

As a point of comparison (not shown), $BlackBox$ takes between 2 (*fanout=1*) to 20 (*fanout=100*) seconds to execute a query where *fanin=1* and around 0.7 seconds when *fanin=100*.

3.10 DISCUSSION AND FUTURE DIRECTIONS

The experiments show that the best strategy is tied to the operator's lineage properties, and that there are orders of magnitude differences between different lineage strategies. Science-oriented lineage systems should seek to identify and exploit operator fanin, fanout, and redundancy properties. This section addresses the generality of our techniques to other

scientific and non-scientific domains, and outlines a number of promising directions for future research.

3.10.1 GENERALITY TO SCIENCE APPLICATIONS

Many scientific applications – particularly sensor-based or image processing applications like environmental monitoring or astronomy – exhibit substantial locality (e.g., average temperature readings within an area) that can be used to define payload, mapping or composite operators. As the experiments show, SubZero can record their lineage with less overhead than operators that only support full lineage.

When locality is not present, as in the genomics benchmark, the optimizer may still be able to find opportunities to record lineage if the constraints are relaxed. An approach that supports lineage at variable granularities is a promising alternative because it can to simplify the process of instrumenting operators for lineage. Developers can define coarser relationships between input and outputs (e.g., specify lineage as a bounding box that may contain inputs that didn't contribute to the output), which is often straight-forward as compared to keeping track of the exact lineage relationship. SubZero could also perform lossy compression by storing lineage at a coarser granularity when resources are limited.

3.10.2 GENERALITY TO DATA APPLICATIONS

The following subsection describes three design principles that apply to provenance management in general data processing systems.

Physical Data Independence

Physical data independence is a well understood topic in the database literature, and it similarly applies to lineage systems. Decoupling the lineage model from how the lineage is represented and encoded is the mechanism that *enables* an optimizer to pick the appropriate lineage strategy based on lineage statistics and query workload. This is analagous to the database query optimizer, which picks the best join execution (e.g., hash join vs sort merge join) depending on the type of query, cardinality estimations, and available indices and views.

Many existing lineage-tracking systems [53, 73, 113] define a fixed storage format and indexing structure that is used for all lineage data in the system. For example, the RAMP [53] system for MapReduce [37] physically co-locates output records with the IDs the records' operator lineage in order to speed up backward lineage queries. This design makes it

challenging to change the encoding or storage schemes and precludes alternative physical layouts that may, for example, be optimized for forward lineage queries.

System Design and Lineage API

The system design as described in Section 3.4 does not make any assumptions about the data-processing system other than that it is an operator-based workflow system. Most modern data-processing systems are operator-based [7, 16, 37, 58, 122] and we believe the design can be re-used for these other workflow systems. In addition, the lineage API provides a simple mechanism, via the *cur_reps* argument passed to the operator, for the runtime system to manage what lineage is written from the operator. This mechanism enables the optimizer and is used to dynamically generate an operator's lineage information during the execution of a lineage query.

Payload Lineage

The payload lineage representation is a simple and flexible approach that can work across data processing systems. For example, the predicate-based lineage in Trio [113] can be implemented by encoding the predicate as the binary payload and executing a filter query based on the predicate inside the *map_p*() method. It can also encode the input record identifiers in RAMP [53].

3.10.3 FURTHER PERFORMANCE OPPORTUNITIES

The results in this chapter have shown the value of a lineage-oriented optimizer. However, as the experiment in Section 3.9.2 mentions, the runtime overhead of a lineage system can often be more than applications are willing to tolerate. More research is needed to further reduce this overhead (in absolute terms) to acceptable levels.

Selective Lineage

In this work, we assume each operator generates, and the runtime stores, *all* of the lineage relationships for the operator. In reality, the application may prioritize a small subset of the results (e.g., new celestial bodies in LSST) over the rest (e.g., empty space or existing stars). The lineage system can significantly reduce its resource overheads by only storing lineage for the prioritized subset.

Composite lineage is a simple application of this insight; it explicitly stores the high priority lineage and represents the rest using a mapping function. However, a general

mechanism to selectively store an operator's lineage information is needed because it is not always possible to define such a mapping function. Exploring how the developer expresses filtering criteria and how the runtime can correctly and efficiently make use of this information is an interesting research direction.

Approximate Lineage

Rather than supporting exact lineage queries, some applications are willing to tolerate lineage results that are imprecise. In other words, results that are a superset of the exact lineage. For example, the LSST astronomers will visually inspect an output cell's lineage as images on the screen and want to use the lineage system to zoom into a relevant portion of the sky.

One approach is for the lineage runtime system to store lineage data using a lossy compression algorithm and ensure that the approximation errors propogate through the workflow. However this approach reduces the storage requirements at the cost of additional runtime overhead for compression.

An alternative is to extend the lineage API to support multi-granularity lineage. Although our region provenance encodings are applicable to a large class of scientific operators, if may be difficult to define mapping functions or the correct *lwrite* calls for complex UDFs. In these cases, the developer may opt to adopt a coarser definition of lineage by specifying coarse regions of input cells. This gives the application control over the amount of approximation, and also reduces the amount of lineage generated by the operator.

3.10.4 LINEAGE SEMANTICS

As hinted in Section 2.1, defining the proper semantics for a given operator, or an entire workflow, can be very difficult because it is application specific and is not meant for "mere-mortals". For example, tracking both explicit (value used to compute result value) and implicit (input used in the control flow) dependencies in the operators is a necessary approach to guarantee reproducibility. On the other hand, if the lineage use case is for manual debugging, then tracking implicit flows may not be necessary.

Rather than implementing provenance semantics and then executing lineage queries, an alterative approach is to specify the semantics of a provenance workload and for the system to suggest different forms of operator semantics that are necessary to accurately execute the provenance queries. This may simplify the need for the developer to both reason about provenance semantics *and* instrument the operators. The key challenge in this approach is

to develop a robust set of provenance query-level semantics that are useful for a large class of applications, yet simple enough to be analyzed.

3.10.5 USING LINEAGE

As evidenced in the experiments, aggregation operators that compute statistics over large subsets of their inputs will result in very large intermediate results (up to the size of the entire input arrays) during the execution of a lineage query results. In these cases, a lineage query will generate a *complete*, but perhaps, *imprecise* result. However, users typically execute lineage queries in order to debug an analysis result and an imprecise may not be useful. This observation suggests that, in order to make lineage metadata *useful* for users, additional algorithms need to be developed to process the lineage query results based on classes of debugging needs.

As a simple example, consider a genomics workflow (Section 3.3.2) that computes the average gene expression per patient. The user is surprised that *patienti*'s average expression levels are very high and queries for that result's lineage. SubZero will accurately return all of *patienti*'s gene expression values, however there can be hundreds of thousands of genes and the user must still comb through them to determine *which* genes are most responsible. In these scenarios, it would be desirable to automatically order subsets of the lineage by an "importance" criteria. Chapter 4 explores this idea further in the context of relational SQL queries.

3.11 CONCLUSION

We introduced SubZero, a scientific-oriented lineage storage and query system that stores a mix of black-box and fine-grained lineage. We explored the design and implementation of an optimization framework that picks the lineage representation on a per-operator basis in order to maximize expected lineage query performance while staying within user constraints. In addition, we developed *region lineage*, which explicitly represents lineage relationships between sets of input and output data elements, along with a number of efficient encoding schemes. For the scientific applications we tested, it can significantly outperform the cell-by-cell lineage that existing systems store.

SubZero is heavily optimized for operators that can deterministically compute lineage from array cell coordinates and small amounts of operator-generated metadata. UDF developers expose lineage relationships by calling the runtime API and/or implementing mapping functions.

Our experiments are run on two application benchmarks – an image processing application in astronomy that exhibits significant lineage locality and data sparsity, and a machine learning application in genomics that does not exhibit locality and operates on dense data. The results suggest that many scientific operators can use our techniques to dramatically reduce the amount of redundant lineage that is generated and stored. This helps improve query performance by up to 10× while using up to 70× less storage space as compared to existing cell-based strategies. The optimizer successfully scales the amount of lineage stored based on application constraints, and can improve the query performance of the genomics benchmark, which is amenable to black-box only strategies.

Alongside these promising results, we find that the amount that normal workflow execution slows down is strongly correlated with the amount of lineage that is generated, and can easily slow the execution by 2×. Further research is needed to understand mechanisms to aggressively constrain the runtime overhead without reverting to a global black box strategy.

In conclusion, we believe SubZero is a valuable initial step to make interactively querying fine-grained lineage a reality for data-intensive scientific applications.

4 Explaining Visualization Outliers

The preceeding chapter describes a mechanism for users to provide outliers in the output of a workflow (e.g., points in the scatterplot output of a visualization workflow) and track their lineage to the input records that generated those outliers. If the visualization is composed of operators that process and output single records, then it is feasible to return the lineage as a table of records. However, most visualizations will aggregate input datasets and render statistical summaries of the data that can be easily visualized. In these cases, each outlier's value can easily depend on thousands or millions of input records. At this scale, naively returning all of the input records is uninformative and techniques to summarize and reduce the lineage are needed.

This chapter describes Scorpion, a hypothesis generation tool that helps explain outliers in the results of SQL aggregation queries. It identifies and summarizes subsets of the input data that are most correlated with the values of user-specified outliers. These summaries can serve as an initial set of explanations for these outliers.

4.1 INTRODUCTION

Data exploration commonly involves *exploratory analysis*, where users try to understand trends and general patterns by fitting models or aggregating data, and then visualizing the results. The resulting visualizations will often reveal *outliers* – aggregate values, or subgroups of points that behave differently than user expectations. For example, a sales trend may rise faster than expected, or the number of system errors spikes during an hour of the day.

When confronted with these outliers, users will naturally want to understand if there are systematic sources of errors, such as a malformed configuration file causing system crashes, present in the data that are responsible for these anomalous values. This form of analysis, which we call *why-analysis*, seeks to uncover these systematic errors by describing the common properties of the input data points or records that caused the outlier outputs. Although a multitude of tools are effective at highlighting and detecting outliers, none

provide *why-analysis* facilities to explain a given set of outputs are outliers.

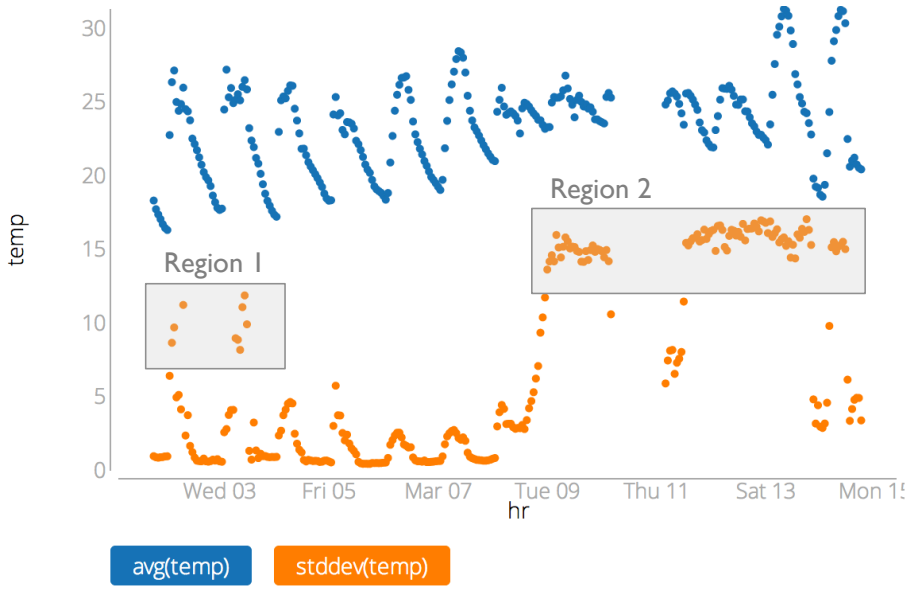


Figure 4-1: Mean and standard deviation of temperature readings from Intel sensor dataset.

For example, Figure 4-1 shows a visualization of data from the Intel Sensor Data Set[1]. Here, each point represents an aggregate (either mean or standard deviation) of data over an hour from 61 sensor motes. Observe that the standard deviation fluctuates heavily (Region 1) and that the temperature stops oscillating (Region 2). Our goal is to describe the properties of the data that generated these highlighted outputs that "explain" why they are outliers. Specifically, we want to find a boolean predicate that when applied to the input data set (before the aggregation is computed), will cause these outliers to look normal, while having minimal effect on the points that the user indicates are normal.

In this case, it turns out that Region 1 is due to sensors near windows that heat up under the sun around noon, and the Region 2 is by another sensor running out of energy (indicated by low voltage) that starts producing erroneous readings. However, these facts are not obvious from the visualization and require manual inspection of the attributes of the readings that contribute to the outliers to determine what is going on. We need tools that can automate analyses to determine e.g., that an outlier value is correlated to the location or voltage of the sensors that contributed to it.

[1] http://db.csail.mit.edu/labdata/labdata.html

4.1.1 PROBLEM OVERVIEW

This problem is fundamentally challenging because a given outlier aggregate may depend on an arbitrary number and combination of input data tuples. Identifying them requires solving the following sub-problems.

Backwards provenance

We need to work backwards from each aggregate point in the outlier set to the input tuples used to compute it (its lineage). In this work we assume that input and output data sets are relations, and that outputs are generated by SQL group-by queries (possibly involving user-defined aggregates) over the input. In general, every output data point may depend on an arbitrary subset of the inputs, and require specialized lineage tracking systems such as SubZero (Chapter 3).

Responsible subset

For each outlier aggregate point, we need a way to determine which subset of its input tuples *most* caused the value to be an outlier. This problem, in particular, is difficult because the naive approach involves iterating over all possible subsets of the input tuples used to compute an outlier aggregate value.

Predicate generation

Ultimately, we want to construct a conjunctive predicate over the input attributes that filter out the points in the responsible subset without removing a large number of other, incidental data points. Thus, the responsible subset must be composed in conjunction with creating the predicates. However, the predicate space is too large to search naively – it is exponential in the dimensionality of the dataset, and in the cardinality of discrete attributes in the dataset.

4.1.2 CONTRIBUTIONS AND CHAPTER ROADMAP

This chapter presents Scorpion, a system we have built to solve the above problems. Scorpion uses *sensitivity analysis* [95] to identify a systematic group of input points that *most influence* the outlier aggregate outputs and generates a predicate that matches the points in the groups. Scorpion's problem formulation and system is designed to work with arbitrary user-defined aggregation functions, albeit slowly for black-box functions. We additionally describe properties shared by many common aggregate functions that enable more efficient algorithms extended from classical regression tree and subspace clustering algorithms.

In Section 4.2, we describe several real applications where the why-analysis problem manifests, such as outlier explanation, cost analysis, fault-analysis, and managing lineage query results.

In order to approach the problem of finding the *most* influential predicate, we need a way to compare the influences of different candidates. Section 4.4 introduces a scoring function that induces a partial ordering over the predicate space and captures the goals described in Section 4.2's use cases.

Section 4.6 describes the design of a general system that searches for influential predicates and a naive algorithm that supports arbitrary aggregation functions. The naive solution iterates through, and computes the score, for all possible predicates. However, the number of possible predicates increases exponentially with the dimensionality of the dataset, and this quickly becomes infeasible for even small datasets.

In response, Sections 4.7-4.10 explore several common aggregation properties (similar to distributive and algebraic OLAP aggregation properties) that enable more efficient algorithms, and develop several such algorithms.

Sections 4.11-4.13 present our experimental setup and results on synthetic and real-world problems. We find that our algorithms are of comparable quality to a naive exhaustive algorithm while taking orders of magnitude less time to run.

4.2 MOTIVATION AND USE CASES

Scorpion is designed to augment data exploration tools with explanatory facilities that find attributes of an input data set correlated with parts of the dataset causing user-perceived outliers. In this section, we first set up the running example used throughout the chapter, then describe several motivating use cases.

4.2.1 SENSOR DATA

Our running example is based on the Intel sensor deployment application described in the Introduction. Consider a data analyst that is exploring a sensor dataset shown in Table 4-2. Each tuple corresponds to a sensor reading, and includes the timestamp, and the values of several sensors. The following query groups the readings by the hour and computes the mean temperature. The left-side columns in Table 4-3 lists the query results.

Tuple id	Time	SensorID	Voltage	Humidity	Temp.
T1	11AM	1	2.74	0.4	34
T2	11AM	2	2.71	0.5	35
T3	11AM	3	2.69	0.4	35
T4	12PM	1	2.71	0.3	35
T5	12PM	2	2.65	0.5	50
T6	12PM	3	2.30	0.4	100
T7	1PM	1	2.71	0.3	35
T8	1PM	2	2.70	0.5	35
T9	1PM	3	2.31	0.5	80

Table 4-2: Example tuples from **sensors** table

Result id	Time	AVG(temp)	Label	v
α_1	11AM	34.6	Hold-out	-
α_2	12PM	61.6	Outlier	$< -1 >$
α_3	1PM	50.0	Outlier	$< -1 >$

Table 4-3: Query results (left) and user annotations (right)

SELECT avg(temp), time (Q1)
FROM sensors GROUP BY time

The analyst thinks that the average temperature at 12PM and 1PM are unexpectedly high and wants to understand why. There are a number of she may want to understand these anomalies:

1. Describe the sensors readings that we can blame for "causing" the anomalies.
2. Describe the readings that *most* "caused" the anomalies.
3. Why are these sensors reporting high temperature?
4. This problem didn't happen yesterday. How did the sensor readings change?

In each of the questions, the analyst is interested in properties of the readings (e.g., sensor id) that most influenced the outlier results. Some of the questions (1 and 2) involve the *degree* of influence, while others involve comparisons between outlier results and normal results (4). Section 4.4 formalizes these notions.

4.2.2 MEDICAL COST ANALYSIS

We are currently working with a major hospital (details anonymized) to help analyze opportunities for cost savings. They observed that amongst a population of cancer patients, the top 15% of patients by cost represented more than 50% of the total dollars spent. Surprisingly these patients were not significantly sicker, and did not have significantly better or worse outcomes than the median-cost patient. Their dataset consisted of a table with one row per patient visit, and 45 columns that describe patient demographics, diagnoses, a break-down of the costs, and other attributes describing the visit. They manually picked and analyzed a handful of dimensions (e.g., type of treatment, type of service) and isolated the source of cost overruns to a large number of additional chemotherapy and radiation treatments given to the most expensive patients. They later found that a small number of doctors were over-prescribing these procedures, which were presumably not necessary because the outcomes didn't improve.

Note that simply finding individually expensive treatments would be insufficient because those treatments may not be related to each other. The hospital is interested in descriptions of high cost areas that can be targeted for cost-cutting and predicates are a form of such descriptions.

4.2.3 FAULT ANALYSIS

Fault analysis is closely related to the previous example. A telecom provider (identify anonymized) we are working with tracks the number of daily fault-related jobs (e.g., a tree branch disables a telephone line) across their network. Analysts view the total number of jobs per day or week and investigate unexpected spikes or upward trends in the total number of faults. They would like to understand the common properties causing the faults to understand which faults to prioritize so so that the number per day is relatively stable.

The dataset contains a table with one row per job network, and columns that describe the type of job, the subregion in the network, and a number of other network related information.

4.2.4 ELECTION CAMPAIGN EXPENSES

In our experiments, we use a campaign expenses dataset [2] that contains all campaign expenses between January 2011 and July 2012 during the 2012 US Presidential Election. In an election that spent an unprecedented $6 billion, many people are interested in where the money was spent. While technically capable users are able to programmatically analyze

[2] http://www.fec.gov/disclosurep/PDownload.do

Notation	Description
D	The input relational table with attributes $attr_1, \cdots, attr_k$
A_{gb}, A_{agg}	Set of attributes referenced in GROUPBY and aggregation clause
$p_i \prec_D p_j$	Result set of p_i is a subset of p_j when applied to D
α	The set of aggregate result tuples, α_i's
g_{α_i}	Tuples in D used to compute α_i e.g., have same GROUPBY key
O, H	Subset of α in outlier and hold-out set, respectively
v_{α_i}	Error description for result α_i

Table 4-4: Notations used

the data, end-users are limited to interacting with pre-made visualizations – a *consumer* role – despite being able to ask valuable domain-specific questions about expense anomalies, simply due to their lack of technical expertise. Scorpion is a step towards bridging this gap by automating common analysis procedures and allowing end-users to perform *analyst* operations.

4.2.5 EXTENDING PROVENANCE FUNCTIONALITY

A key provenance use case is to trace an anomalous result backward through a workflow to the inputs that directly affected that result. A user may want to perform this action when she sees an anomalous output value. Unfortunately, when tracing the inputs of an aggregate result, the existing provenance system will flag a significant portion of the dataset as the provenance [33]. Although this is technically correct, the results are not *precise*. Scorpion can reduce the provenance of aggregate operators to a small set of influential inputs that is easier for an analyst to digest.

4.3 PROBLEM SETUP

This section introduces notations that will be used in the rest of the chapter, summarized in Table 4-4.

Consider a single relation D with attributes $A = attr_1, .., attr_k$. Let Q be a non-nested group-by SQL query grouped by attributes $A_{gb} \subset A$, with a single aggregate function, $agg()$, that computes a result using aggregate attributes $A_{agg} \subset A$ from each tuple, where $A_{agg} \cap A_{gb} = \varnothing$. Finally, let $A_{rest} = A - A_{gb} - A_{agg}$ be the attributes not involved with the aggregate function nor the group by that are used to construct the explanations.

For example, Q1 contains a single group-by attribute $A_{gb} = \{time\}$, and an aggregate attribute $A_{agg} = \{temp\}$. The user is interested in combinations of $A_{rest} =$

$\{SensorID, Voltage\}$ values that are responsible for the anomalous average temperatures.

Scorpion outputs the predicate that most influences a set of output results. A predicate p is a conjunction of range clauses over the continuous attributes and set containment clauses over the discrete attributes, where each attribute is present in at most one clause. $\neg p$ is the negation of p, and P_A is the space of all possible predicates over the attributes in A. Let $p(D) = \sigma_p D \subseteq D$ be the set of tuples in D that satisfy p. A predicate p_i is contained in p_j with respect to a dataset D if the tuples in D that satisfy p_i are a subset of those satisfying p_j:

$$p_i \prec_D p_j \leftrightarrow p_i(D) \subset p_j(D)$$

Let the query generate n aggregate result tuples $\alpha = \{\alpha_1, .., \alpha_n\}$, and the lineage of a result α_i be denoted $l_i \subseteq D$ [3]. The output attribute $\alpha_i.res = agg(\pi_{A_{agg}} l_i)$ is the result of the aggregate function computed over the projected attributes, A_{agg}, of the tuples in l_i.

Let $O = \{o_1, .., o_{n_s} | o_i \in \alpha\}$ be a subset of the results that the user flags as outliers, and $H = \{h_1, \cdots, h_{n_h} | h_i \in \alpha\}$ be a hold-out set of the results that the user finds normal. O and H are typically specified through a visualization interface, and $H \cap O = \varnothing$. Let $g_{\mathcal{X}} = \cup_{x \in \mathcal{X}} g_x | \mathcal{X} \subseteq \alpha$ be shorthand for the lineage of a subset of the results, $\mathcal{X}$. For example, g_O denotes the lineage of the outliers.

The user can also specify how the outlier result looks wrong by specifying. For a result o, she can specify an error description $v_o \in \{high, low, wrong, eq_i\}$ for when o is too high and its value should be decreased as much as possible ($high$), too low and its value should be increased(low), simply wrong and its value should change in any direction ($wrong$), or should be equal to i (eq_i). Let $V = \{v_{o_i} | o_i \in O\}$, be the set of error descriptions of all of the outlier results.

4.4 FORMALIZING INFLUENCE

Scorpion seeks to find a predicate over an input dataset that most influences a user selected set of query outputs. In order to reason about this problem, we must define a partial ordering of the predicate space so that we can distinguish preferable predicates from non-preferable ones.

This section introduces the influence scoring function $inf_{agg}(\bullet)$ that defines such a partial ordering. We will build up its argument list starting from the most basic definition that handles a single outlier result o whose value is too high. We then increase the function's

[3] The lineage semantics are the same as those in Panda [56] e.g., the subset of input tuples that satisfy the query's selection clauses and whose A_{gb} values are equal to that of α_i.

complexity by adding support for: an error type v_o; a hold-out result h; parameters that control the trade-off between "fixing" the outlier, the result predicate' cardinality, and the amount the hold-out is perturbed. The final version handles multiple outlier and hold-out results.

4.4.1 BASIC DEFINITION

Our notion of influence is derived from sensitivity analysis [94], which computes the sensitivity of a model to its inputs. Given a function $y = f(x_1, \cdots, x_n)$, the influence of x_i is defined by the amount the output changes given a change in the x_i (the partial derivative) $\frac{\Delta y}{\Delta x_i}$.

In our context, the model is an aggregation function $agg()$ that takes a *set* of tuples such as l_o as input, and outputs a result o. A predicate p's influence on o depends the on the difference between the original result $o.res$ and the updated output after deleting $p(l_o)$ from l_o. Note the analogy to Δy in the partial derivative[4].

$$\Delta o = \Delta_{agg}(o, p) = agg(l_o) - agg(\neg p(l_o))$$

As such, the trivial solution $p = True$ would maximize this score for aggregation functions such as $COUNT$. Thus we add a regularization term $\Delta l_o = |p(l_o)|$ that represents the change in the aggregation function input l_o, and redefine influence as the ratio between Δo and Δl_o. [5].

$$inf_{agg}(o, p) = \frac{\Delta o}{\Delta l_o} = \frac{\Delta_{agg}(o, p)}{|p(l_o)|}$$

For example, suppose the individual influences of each tuple in $l_{\alpha_2} = \{T4, T5, T6\}$, from Tables 4-2 and 4-3. Based on the above definition, removing T4 from the input group increases the output by 13.39, thus T4 have an influence of $inf_{AVG}(\alpha_2, \{T4\}) = \frac{61.6-75}{1} = -13.39$. In contrast, T6 has an influence of 19.2. Given this definition, T6 is the most influential tuple, which makes sense, because T6.temp increases the average the most, so removing it would most reduce the output.

The reason Scorpion defines influence in the context of predicates rather than individual or sets of tuples is because individual tuples only exist within the lineage of a single result tuple, whereas predicates are applicable to the lineage of multiple results. We now augment inf with additional arguments to support other user inputs.

[4] Alternative formulations, e.g., perturbing input tuple values rather than deleting inputs tuples, are also possible but not explored here.

[5] This definition closely resembles the discrete derivative of $agg()$.

4.4.2 ERROR DESCRIPTION

The previous formulation does not take into account the error descriptions, i.e., whether the outliers are too high or too low. For example, if the user thinks that the average temperature was too *low*, then removing T6 would, contrary to the user's desire, further decrease the mean temperature. We support this by modifying the definition of Δ to also depend on v_o:

$$\Delta_{agg}(o, p, v_o) = \begin{cases} agg(l_o) - agg(\neg p(l_o)) & \text{if } v_o = high \\ agg(\neg p(l_o)) - agg(l_o) & \text{if } v_o = low \\ |agg(l_o) - agg(\neg p(l_o))| & \text{if } v_o = wrong \\ 1 - \frac{1+|val - agg(\neg p(l_o))|}{1+|val - agg(l_o)|} & \text{if } v_o = eq_{val} \end{cases}$$

When $v_o = high$, the Δ function is identical to the previous definition. However if the user believes that the outlier is too low or wrong, then simply negating or taking the absolute value of Δo is sufficient to capture that notion. If the user states that the output value should be val (e.g., $v_o = eq_{val}$), then we compute the absolute euclidian distances between val and the updated as well as original output values. The ratio of these two distances represents how close the o's new value to val as compared with the original. We use add-one smoothing to deal with the case that o is already equal to val.

To complete our modification, we extend the influence function to propogate the error description to the Δ function:

$$inf_{agg}(o, p, v_o) = \frac{\Delta_{agg}(o, p, v_o)}{|p(l_o)|}$$

4.4.3 C HYPERPARAMETER

If the user specifies that an outlier result is "too high", how aggressively should Scorpion attempt to reduce its value? For example, let us compare $p_1 = \text{voltage} < 2.5$, which matches $\{T6, T9\}$, and $p_2 = \text{voltage} \leq 2.65$, which matches $\{T5, T6, T9\}$. Both predicates describe anomalous temperatures higher than 35^o, however p_1 matches the very high temperature readings, while p_2 matches all readings above 35^o. Since both predicates seem plausible our influence function should have a mechanism to let the user prefer p_1 or p_2.

To support this, we modify the influence functions to accept an extra c parameter, which is used as the exponent of the denominator in the influence function:

$$inf_{agg}(o, p, v_o, c) = \frac{\Delta_{agg}(o, p, v_o)}{|p(l_o)|^c}$$

The exponent $c \geq 0$ controls the trade-off between the importance of keeping the size of $p(l_o)$ small and maximizing the desired change in the output. In effect, when a user specifies that an outlier result is too high, c controls how aggressively Scorpion should reduce the result. For example, when $c = 0$, Scorpion will reduce the aggregate result without regard to the number of tuples that are used, producing predicates that select many tuples. Increasing c places more emphasis on finding a smaller set of tuples that have more "bang for the buck", producing much more selective predicates.

As a concrete example, Figure 4-14 illustrates a simple 2D predicate space where each point represents a tuple, the color represents $A.agg$ and varies from grey (low), medium (orange), to high (red). The user computes the average of all of the record values and believes the result is too high. The rectangle is a predicate that contains the influential subset. As c increases, the rectangle shrinks to focus on the highest value tuples at the expense of less total influence on the aggregation result.

4.4.4 HOLD-OUT RESULT

As mentioned above, a hold-out result h is a result that p should not influence, so p should be penalized if it influences the hold-out results in any way. Unfortunately, there may not exist a predicate that selectively influences the outliers without modifying the hold-outs and we will need a way to manage these competing goals. To this end, we extend the influence function to manage this trade-off using a parameter λ:

$$inf_{agg}(o, h, p, v_o, c) = \lambda \times inf_{agg}(o, p, v_o, c) - (1 - \lambda) \times |inf_{agg}(h, p, 0)| \tag{4.1}$$

The absolute value of $inf_{agg}(h, p)$ serves to penalize *any* perturbation of the hold-out result.

Note that our treatment of h could be uniformly supported by viewing h as a special case of an outlier whose error description is $eq_{h.res}$. However, we distinguish between outlier and hold-out results both for clarity in the text, and so that different weights (λ) can be explicitly applied to the outliers and hold-outs.

4.4.5 MULTIPLE RESULTS

The user will often select multiple outlier results O and hold-out results H. We extend the influence function to multiple result by computing the average of the outlier results and

penalizing the maximum perturbation of the hold-out results:

$$inf_{agg}(O, H, p, V, c) = \lambda \times \underset{o \in O}{\text{avg}}\, inf_{agg}(o, p, v_o, c) - (1 - \lambda) \times \max_{h \in H} |inf_{agg}(h, p, 0)|$$

We chose *avg* in order to balance the desire to influence a substantial subset of the outliers[6] with the reality that there may not exist a single predicate that influences all outliers[7] In addition, average has attractive computational properties (e.g., it is smooth and can be incrementally computed) that robust functions such as *median* do not support. That being said, other functions such as *median* or *quartile* are perfectly valid.

We chose *max* in order to provide a hard cap on the amount that a predicate can influence *any* hold-out result. Alternatively, we could use the top decile, which may provide more robust support if the client unknowingly chooses a few unlucky hold-out values.

4.4.6 NOTATIONAL SHORTHANDS

The rest of the chapter uses the following short-hands when the intent is clear from the context.

$$inf(p) = inf_{agg}(O, H, p, V, c)$$
$$\Delta(p) = \Delta_{agg}(o, p)$$

Functions are also extended to interpret a single tuple as a single element set:

$$inf(t) = inf(\{t\})$$
$$\Delta(t) = \Delta(\{t\})$$

4.4.7 INFLUENTIAL PREDICATES PROBLEM

We can now introduce the **Influential Predicates (IP) Problem**: Given a select-project-group-by SQL query Q, and client inputs O, H, V, λ and c, find the predicate, p^*, from the

[6] *max* may degenerate towards influencing a single result.

[7] *min* cannot distingiush between predicates that do not influence all of the outliers.

set of all possible predicates, $P_{A_{rest}}$, that has the maximum influence:

$$p^* = \arg\max_{p \in P_{A_{rest}}} inf(p) \tag{4.2}$$

Why is This Problem Hard?

Section 4.2 motivated why this problem is useful, but it is not immediately obvious why this problem should be difficult. For example, if the user thinks the average temperature is too high, why not simply return the readings with the highest temperature? We now illustrate some reasons that make the *IP* problem difficult. The rest of this chapter will explore efficient solutions to this problem.

Non-independence Scorpion needs to consider how combinations of input tuples affect the outlier results, which depends on properties of the aggregate function. In the worst case, Scorpion cannot predict how combinations of input tuples interact with each other, and needs to evaluate all possible predicates (exponential in the number of and cardinalities of attributes). Section 4.7.2 explores a class of aggregation functions where this restriction can be relaxed.

Working with Predicates Scorpion provides the user with understandable explanations of anomalies in the data by returning *predicates* rather than individual tuples. Thus, Scorpion must find tuples within bounding boxes defined by predicates, rather than arbitrary combinations of tuples. In the example above, it may be tempting to find the top-k highest temperature readings and construct a predicate from the minimum bounding box that contains those readings. However, it is unclear how many of the top readings should be used and what the right cut-off should be. In fact, the top readings may have no relation with each other and the resulting predicate may be non-influential because it primarily contains a number of normal or low temperature

Query-Dependent The influence of a predicate relies on statistics of the tuples in addition to their individual influences, and the specific statistic depends on the particular aggregate function. For example, AVG depends on both the values and density of tuples, while $COUNT$ only depends on the density.

Hold-outs In the presence of a hold-out set, simple hill-climbing algorithms may not work because a predicate that influences the outliers may also influence the hold-out results. The

non-convexity of the influence function combined with the size of the problem space makes the problem particularly challenging and necessitates strong assumptions and/or heuristics.

4.5 ASSUMPTIONS

Recall that our goal is to find subsets of an input dataset (in the form of a predicate) whose removal appears to fix the values of result outliers. To evaluate different candidate solutions, we defined a distance function between the original result values and the updated results that describes the amount the outliers have been fixed. A necessary condition to evaluate the distance function is the ability to unambiguously compare each original result value with the updated value.

Unfortunately, this condition does not hold for arbitrary SQL functions. To simplify our reasoning, we made three assumptions about the structure of the SQL query – the query is a group-by aggregation, does not contain subqueries, and does not contain joins. The rest of this section explains our rationale for each of these restrictions.

Group-by Assumption

The group-by restriction is necessary because it enables the aggregation operation that forms the basis of our problem. Without an aggregation operator, each result is trivially dependent on a single input record (in a single relation query). When the user specifies the outlier set, it is analagous to labelling individual points in a supervised learning problem and we can use a standard rule-based learning algorithm such as a decision tree [91] to describe the outliers.

Subquery Assumption

We disallow subqueries because it allows queries where the distance function cannot be unambiguously evaluated. To see why this is valuable, consider the following nested query that Scorpion does not handle:

SELECT sum_b, sum(a) as sum_a (Q2)
FROM (
    SELECT a, sum(b) as sum_b
    FROM $T_{example}$
    GROUP BY a) as T_{interm}
GROUP BY sum_b

id	a	b	c
0	0	1	1
1	0	2	0
2	1	3	0
3	1	0	0
4	2	2	1
5	2	1	0

(a) $T_{example}$

id	sum_b	sum_a
r_0	3	3

(b) Result of Q2($T_{example}$)

id	sum_b	sum_a
r'_0	1	2
r'_1	2	0
r'_2	3	1

(c) Result of Q2($\sigma_{c\neq 1} T_{example}$)

Figure 4-5: Tables in example problem to show that IP problem is ill-defined under Q2

The subquery in Q2 partitions the data and produces three tuples $\{(0,3),(1,3),(2,3)\}$. The outer query then groups the data on the second attribute to compute the final results in Table ??. In contrast, if the input table is filtered as $\sigma_{c\neq 1} T_{example}$, then the subquery will produce three intermediate tuples $\{(0,2),(1,3),(2,1)\}$, and the outer query will produce the results in Table 4-5c. Since the updated query generates more results whose lineage overlaps with r_0, it is ambiguous which updated result should be used to compare against r_0. This restriction help us sidestep this ambiguity.

Join Assumption

Our restriction on joins is for both convenience and efficiency. Efficient procedures to "refresh" output results given changes in the input dataset have been well studied by Ikeda et. al [54, 55]. Thus, Scorpion technically supports arbitrary join queries using its naive algorithm and we do not consider joins to keep the text simple.

A second concern is that it makes designing efficient search procedures difficult because an input tuple may both contribute several times to a single result, and may contribute to multiple results. The latter suggests that algorithms that treat each l_{o_i} independently may not be safe because we need to track tuples whose contributions span multiple o_i's. For this reason, we make the simplifying assumption and deal with joins as a future research direction.

4.6 BASIC ARCHITECTURE

This section outlines the Scorpion system architecture we have developed to solve the problem of finding influential predicates defined in the previous section and describes naive implementations of the main system components. These implementations do not assume

anything about the aggregates so can be used on arbitrary user defined aggregates to find the most influential predicate. We then explain why these implementations are inefficient.

4.6.1 SCORPION ARCHITECTURE

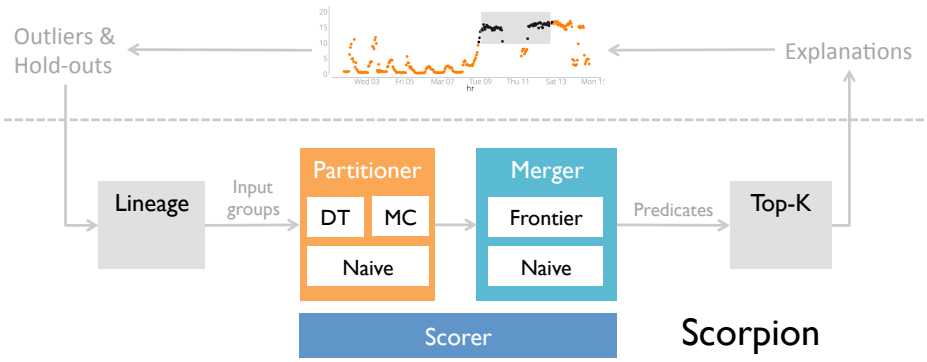


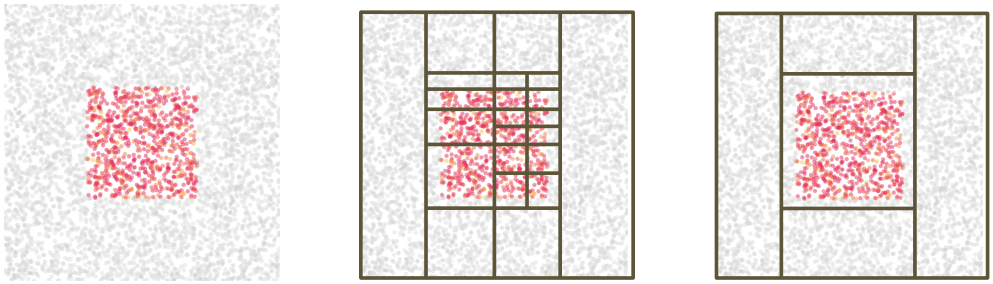
Figure 4-6: Scorpion architecture

Scorpion is implemented as part of an end-to-end data exploration tool (Figure 4-6). Users can select databases and execute SQL aggregation queries whose results are visualized as charts (Figure 4-1 shows a screenshot). Users can select arbitrary results, label them as outliers or hold-outs, specify attributes that should be ignored during the predicate search, and send the query to the Scorpion backend. Users can click through the result explanations and plot the updated output with the outlier inputs removed from the SQL query.

Scorpion first uses the *Lineage* component to compute the lineage of the labeled results. In this work, the queries are group-by queries over a single table, so computing the lineage is straightforward. More complex relationships can be established using relational provenance techniques [33] or a full-fledged lineage system such as SubZero.

The lineage, along with the original inputs, are passed to the *Partitioner*, which chooses the appropriate partitioning algorithm based on the properties of the aggregate. The algorithm generates a ranked list of predicates, where each predicate is tagged with a score representing its estimated influence. For example, consider the 2D dataset illustrated in Figure 4-7a, where each point represents an input tuple and a darker color means higher influence. Figure 4-7b illustrates a possible partitioning of the dataset, where each partition is a predicate. The partitioning algorithms often over-partition the dataset (i.e., each predicate contains a subset of the optimal predicate) so Scorpion executes a merging phase (*Merger*), which greedily merges similar predicates as long as it increases the influence (Figure 4-7c).

The *Partitioner* and *Merger* send candidate predicates to the *Scorer*, which computes the influence as defined in the previous section. Computing the Δ values dominates the cost



(a) Input dataset (b) Output of *Partitioner*. (c) Output of *Merger*

Figure 4-7: Each point represents a tuple. Red color means higher influence.

of this component because it needs to remove the tuples that match the predicate from each result's lineage, then rerun the aggregate on the updated lineage. This cost can be very high if a result is computed from a large set of input tuples, or if the aggregation function makes multiple passes over the data. Section 4.7.1 describes a class of aggregation functions that can reduce these costs.

Finally, the top ranking predicate is returned to the visualization and shown to the user. We now present basic implementations of the partitioning and merging components.

4.6.2 NAIVE PARTITIONER (**naive**)

For an arbitrary aggregate function without nice properties, it is difficult to improve beyond an exhaustive algorithm that enumerates and evaluates all possible predicates. This is because the influence of a given tuple may depend on the other tuples in the outlier set, so a simple greedy algorithm will not work. The NAIVE algorithm first defines all distinct single-attribute clauses, then enumerates all conjunctions of up to one clause from each attribute. The clauses over a discrete attribute, A_i, are of the form, "$\mathrm{A_i}$ in $(\cdots)$" where the $\cdots$ is replaced with all possible combinations of the attribute's distinct values. Clauses over continuous attributes are constructed by splitting the attribute's domain into a fixed number of equi-sized ranges, and enumerating all combinations of consecutive ranges. *NAIVE* computes the influence of each predicate by sending it to the *Scorer*, and returns the most influential predicate.

This algorithm is inefficient because the number of single-attribute clauses increases exponentially (quadratically) as the cardinality of the discrete (continuous) attribute increases. Additionally, the space of possible conjunctions is exponential with the number of attributes. The combination of the two issues makes the problem untenable for even small datasets. While the user can bound this search by specifying a maximum number of clauses allowed in a predicate, enumerating all of the predicates is still prohibitive.

4.6.3 BASIC MERGER

The *Merger* takes as input a list of predicates ranked by an internal score, iteratively merges subsets of the predicates, and returns the resulting list. Two predicates are merged by computing the minimum bounding box of the continuous attributes and the union of the values for each discrete attribute. The basic implementation repeatedly expands the existing predicates in decreasing order of their scores. Each predicate is expanded by greedily merging it with adjacent predicates until the resulting influence does not increase.

This implementation suffers from multiple performance-related issues if the aggregate is treated as a black-box. Each iteration calls the *Scorer* on the merged result of every pair of adjacent predicates, but may only successfully merge a single pair. In addition, it is susceptible to the curse of dimensionality, because the number of neighbors increases exponentially with the number of attributes in the dataset. Section 4.9 explores optimizations that address these issues.

The next section will describe several aggregate operator properties that enable more efficient algorithm implementations.

4.7 QUERY AND AGGREGATION PROPERTIES

To compute results in a managable time, algorithms need to efficiently estimate a predicate's influence, and prune the space of predicates. These types of optimizations depend on making stronger assumptions about the aggregation function. This subsection describes several properties that, when satisfied by an aggregation function, enables more efficient search algorithms. Developers only need to specify these properties for their aggregation functions once, and they are transparent to the end-user.

4.7.1 INCREMENTALLY REMOVABLE

The *Scorer* is extensively called from all of our algorithms, so reducing its cost is imperative. Its most expensive operation is computing the Δ value by recomputing the aggregrate function on the filtered input dataset. If the candidate predicate p does not match many tuples, then $|D| \approx |\neg p(D)|$ and the cost is nearly equivalent to re-running the query on the entire dataset. It would be desireable to *incrementally* compute the aggregate result by only examining the tuples that match p.

Example

As a concrete example, consider SUM over the values $D = \{1, 2, 3, 4, 5\}$ and the predicate $p = (value \geq 4)$. To compute the updated result, we would execute:

$$SUM(\neg p(D)) = SUM(\{1, 2, 3\}) = 6$$

Alternatively, we know that the updated value can be incrementally computed:

$$\begin{aligned} SUM(\neg p(D)) = \\ SUM(D - \{4, 5\}) = \\ SUM(D) - SUM(\{4, 5\}) = \\ SUM(D) - 9 = 6 \end{aligned}$$

Since the user's original query has already computed $SUM(D)$, we only need to compute $SUM(p(D))$. We call this ability to incrementally compute $agg(\neg p(D))$ as the *incrementally removable* property.

Definition

In general, a computation is *incrementally removable* if the updated result of removing a subset, s, from the inputs, D, can be computed by only reading s. It also turns out that computing influence of an aggregate is incrementally removable as long as the aggregate itself is incrementally removable.

Formally, an aggregate function, agg, is incrementally removable if it can be decomposed into functions $state$, $update$, $remove$ and $recover$, such that:

$$\begin{aligned} state(D) &\rightarrow m_D \\ update(m_{S_1}, \cdots, m_{S_n}) &\rightarrow m_{\cup_{i \in [1,n]} S_i} \\ remove(m_D, m_{S_1}) &\rightarrow m_{D - S_1} \\ agg(D) &= recover(m_D) \end{aligned}$$

Where D is the original dataset and $S_1 \cdots S_n$ are non-overlapping subsets of D to remove. $state$ computes a constant sized summary tuple m that summarizes the aggregation operation, $update$ combines n summary tuples into one, $remove$ computes the summary

tuple of removing S_1 from D, and *recover* recomputes the aggregate result from the summary tuple.

The *Scorer* uses this property to compute and cache $state(D)$, and re-used the cached result to evaluate subsequent Δ values. A predicate's influence is computed by removing the predicate's tuple from m_D, and calling *recover* on the result. Section 4.9 describes a case where the *Merger* can use summary tuples to approximate influence scores without calling the *Scorer* at all.

Application

This definition is very related to the concept of distributive and algebraic functions in OLAP cubes [42]. These are functions where a sub-aggregate can be stored as a constant-sized summary, and the summaries can be composed to compute the complete aggregate. Whereas OLAP cubes are use this property to *compose* larger aggregates from smaller ones, *incrementally removable* functions want to *remove* sub-aggregates from a larger aggergates.

Despite the similarities, not all distributive or algebraic are incrementally removable. For example, it is not in general possible to re-compute MIN or MAX after removing an arbitrary subset of inputs without knowledge of the full dataset. Similarly, robust statistics such as $MEDIAN$ and $MODE$ are not incrementally-removable. In general, arithmetic expressions derived from $COUNT$ and SUM such as AVG, $STDDEV$, $VARIANCE$ and $LINEAR\_CORRELATION$ are incrementally removable.

A developer implements the procedures *state*, *update*, *remove* and *recover* to make an aggregation function incrementally removable. For example, AVG is augmented as:

$$
\begin{aligned}
AVG.state(D) &= (SUM(D), |D|) \\
AVG.update(m_1, \cdots, m_n) &= (\textstyle\sum_{i\in[1,n]} m_i[0], \sum_{i\in[i,n]} m_i[1]) \\
AVG.remove(m_1, m_2) &= (m_1[0] - m_2[0], m_1[1] - m_2[0]) \\
AVG.recover(m) &= m[0]/m[1]
\end{aligned}
$$

4.7.2 INDEPENDENT

The IP problem is non-trivial because combinations of input tuples can potentially influence a user-defined aggregate's result in arbitrary ways. The *independence* property allows Scorpion to assume that the input tuples influence the aggregate result independently. For a function agg to be independent, it must satisfy the two requirements described below.

Definition

Let $t_1 \leq \cdots \leq t_n$ such that $\forall_{i \in [1,n-1]} inf_{agg}(o, t_i) \leq inf_{agg}(o, t_{i+1})$ be an ordering of tuples the lineage l_o by their influence on the result o. Let T be a set of tuples, then agg must first satisfy the following:

$$t_a < t_b \rightarrow inf_{agg}(T \cup \{t_a\}) < inf_{agg}(T \cup \{t_b\}) \tag{R1}$$

This requirement states that the influence of a set of tuples strictly depends on the influences of the individual tuples without regard to the tuples in T (they do not interact with t_a or t_b). For example, $t_a = 100$ increases the result of AVG more than $t_b = 50$, independent of the existing average value.

In addition, agg must satisify a second condition. Let T_1 and T_2 be two subsets of the input dataset:

$$\frac{\text{avg}_{\,t \in T_1}\, inf_{agg}(t)}{\text{avg}_{\,t \in T_2}\, inf_{agg}(t)} \propto \frac{inf_{agg}(T_1)}{inf_{agg}(T_2)} \tag{R2}$$

This states that the relative differences in the influence of two sets of tuples T_1 and T_2 is proportional to the average influences of the individual tuples in each set.

These requirements point towards a greedy strategy to find the most influential set of tuples for independent aggregates. Assume that the user provided a single suspicious result and no hold-outs. The algorithm first sorts the tuples by influence and then incrementally adds the most influential tuple to the candidiate set until the influence of the set does not increase further. At this point we can construct a predicate using a standard rule-learning algorithm [91]. This algorithm is guaranteed to find the optimal tuple set, though not necessarily the optimal predicate.

While this sounds promising, the requirement is difficult to reason about because it depends on internal details of the aggregation function and the parameters of our influence definition. For example, factors such as the cardinality of the predicate and the presence of hold-out results affect whether this proprety holds. For this reason, we modify the requirement to depend on agg, rather than inf_{agg}. The developer specifies that an operator is independent by setting the attribute, $agg.independent = True$.

Example

Nearly all non-robust statistical functions satisfy requirement R1, however only normalized aggregates such as *AVG*, *STDDEV*, and higher moments of centrality satisfy R2. Functions such as $COUNT$ and SUM do not because their results depend on the cardinality of the dataset. Take the SUM function for example:

$$\begin{gathered}\underset{t\in\{2,2\}}{\text{avg}}\ \Delta_{SUM}(t) < \underset{t\in\{3\}}{\text{avg}}\ \Delta_{SUM}(t) \\ \not\Rightarrow \\ \Delta_{SUM}(\{2,2\}) < \Delta_{SUM}(\{3\})\end{gathered}$$

Although each value in $\{2,2\}$ has a smaller Δ than each value in $\{3\}$, the former set, in aggregate, has a larger Δ_{agg} value because its total SUM is $4 > SUM(\{3\})$.

Section 4.8.1 describes the DT partitioning algorithm that is optimized for this property.

4.7.3 ANTI-MONOTONIC

The anti-monotonic property is used to prune the search space of predicates. In general, a property is anti-monotone if, whenever a set of tuples s violates the property, so does any subset of s. In our context, an operator is anti-monotonic if the amount that a predicate p influences the aggregate result $inf(o,p)$ is greater than or equal to the influence of any predicate contained within p:

$$p' \prec p \iff inf(p') \leq inf(p) \tag{R3}$$

In other words, if p is non-influential, then none of the predicates contained in p can be influential, and p can be pruned. For example, if D is a set of non-negative values, then $SUM(D) > SUM(s)\ \forall_{s\subseteq D}$. This is similar to the downward clouser property used in the apriori algorithm [6] in association rule mining, and algorithms in subspace clustering [5]. Note that the property only holds if the data does not contain negative values.

Similar to the *independence* property, it is non-trivial to determine anti-monotonicity at the influence level. Thus, developers only specify whether *agg* obeys this property by defining a boolean function $agg.check\_antimonotone(D)$, that returns $True$ if D satisfies any required constraints, and $False$ otherwise. For example:

$$COUNT.check\_antimonotone(D) = True$$
$$MAX.check\_antimonotone(D) = True$$
$$SUM.check\_antimonotone(D) = \forall_{d \in D} \ \ d \geq 0$$

Section 4.8.2 describes the *MC* partitioning algorithm that is optimized for this property.

4.8 PARTITIONING ALGORITHMS

While the general IP problem is exponential, the properties presented in the previous section enable several more efficient partitioning and merging algorithms. In this section, we describe a top-down partitioning algorithm that takes advantage of operator independence and a bottom-up algorithm for independent, anti-monotonic aggregates.

A benefit of these partitioning algorithms is that they largely execute independently of the c.

4.8.1 DECISION TREE (**DT**) PARTITIONER

DT is a top-down partitioning algorithm for independent aggregates. It is based on the intuition that the Δ will not significantly change when tuples with similar influence are combined together. Correspondingly, *DT* generates predicates where the lineage of a result α_i that satisfy a predicate have similar influence. The *Merger* then greedily merges adjacent predicates with similar influence to produce the final predicates.

DT recursively splits the attribute space to create a set of predicates. Because the outlier groups are different than hold-out groups, we partition these groups separately, resulting in a set of outlier predicates and hold-out predicates. These are combined into a set of predicates that differentiates ones that only influence outlier results from those that also influence hold-out results. We first describe the partitioning algorithm for a single input group, then for a set of outlier input groups (or hold-out input groups), and finally how to combine outlier and hold-out partitionings.

Single α Recursive Partitioning

The recursive partitioner takes a single lineage set, aggregate, and error description (for outliers) as input, and returns a partitioning[8] such that the variance of the influence of individual tuples within a partition is less than a threshold. Our algorithm is based on regression tree algorithms, so we first explain a typical regression tree algorithm before describing our differences.

Regression trees [20] are the continuous counterpart to decision trees and used to predict a continuous attribute rather than a categorical attribute. In the general formulation, the tree begins with all data in a partition. The algorithm fits a constant or linear formula to the tuples in the partition, and computes the formula's error (typically standard error or sum error). If the error metric or number of tuples in the partition are below their respective thresholds, then the algorithm stops. Otherwise, the algorithm computes the best (attribute, value) pair to split the partition so that the resulting *child partitions* will minimize the error metric, and recursively calls the algorithm on the children.

Our approach re-uses the regression tree framework to minimize the distribution of influence values within a given partition. In our formulation, we set tuple influence as the target attribute, fit a constant formula, define error metric as the standard error, and only consider attribute bisections rather than arbitrary split points.

Stopping Condition

Our key insight is that that partitions containing influential tuples should be more accurate than non-influential partitions, thus the error metric threshold can be relaxed for partitions that don't contain any influential tuples. This way, large perturbations in non-influential partitions will not trigger non-productive splitting.

The error threshold value is based on the maximum influence in a partition, inf_{max}, and the upper, inf_u, and lower, inf_l, bounds of the influence values in the dataset. The threshold can be computed via any function that decreases from a maximum to a minimum threshold value as inf_{max} approaches inf_u. Scorpion computes the threshold as:

[8] Partitions and predicates are interchangeable, however the term partition is more natural when discussing space partitioning algorithms such as those in this section.

$$threshold = \omega * (inf_u - inf_l)$$
$$\omega = min(\tau_{min} + s * (inf_u - inf_{max}), \tau_{max})$$
$$s = \frac{\tau_{min} - \tau_{max}}{(1-p) * inf_u - p * inf_l}$$

Where ω is the multiplicative error as depicted in Figure 4-8, s is the slope of the downward curve, $p = 0.5$ is the inflection point when the threshold starts to decrease, and τ_{max} and τ_{min} are the maximum and minimum threshold values. In our experiments, we set τ_{max} and τ_{min} to 0.05 and 0.001, respectively.

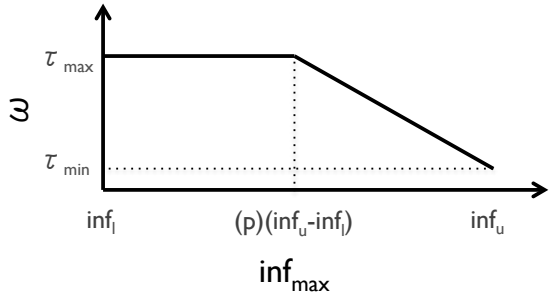


Figure 4-8: Threshold function curve as inf_{max} varies

Sampling

The previous algorithm still needs to compute the influence on all of the input tuples. To reduce this cost, we exploit the observation that the influential tuples should be clustered together (since Scorpion searches for predicates), and sample the data in order to avoid processing all non-influential tuples. The algorithm uses an additional parameter, ϵ, that represents the maximum percentage of the dataset that contains outlier (thus influential) tuples. The system initially estimates a sampling rate, $samp\_rate$, such that a sample from D of size $samp\_rate * |D|$ will contain high influence tuples with high probability ($\geq 95\%$):

$$sample\_rate = min(\{sr | sr \in [0,1] \wedge 1 - (1-\epsilon)^{sr*|D|} \geq 0.95\})$$

Scorpion initially uniformly samples the data, however after computing the influences of the tuples in the sample, there is information about the distribution of influences. We use this when splitting a partition to determine the sampling rate for the sub-partitions. In particular, we stratify samples based on the total relative influences of the samples that fall

into each sub-partition. In this way, the algorithm pays more attention to higher influence regions.

To illustrate, let D be partitioned by the predicate p into $D_1 = p(D)$ and $D_2 = \neg p(D)$, and $S \subset D$ be the sample with sampling rate $samp\_rate$. We use the sample to estimate D_1's (and similarly D_2's) total influence:

$$sum\_inf_{D_1} = \sum_{t \in p(S)} inf(t)$$

The sampling rates are computed as:

$$samp\_rate_{D_1} = \frac{sum\_inf_{D_1}}{sum\_inf_{D_1} + sum\_inf_{D_2}} * \frac{|S|}{|D_1|}$$
$$samp\_rate_{D_2} = \frac{sum\_inf_{D_2}}{sum\_inf_{D_1} + sum\_inf_{D_2}} * \frac{|S|}{|D_2|}$$

Multi-α Recursive Partitioning

When there are multiple l_{α_i} sets, DT needs to find a single partitioning across the lineage of each α_is. To do this, the algorithm separately evaluates a given partition on each l_{α_i}, and merges the error metrics to make consistent termination and split decisions.

For example, DT makes a split decision by combining the error metrics computed for each candidate attribute. For an attribute $attr$, we compute its combined error as $metric_{attr} = max(metric^i_{attr} | i \in [0, |R|])$, where $metric^i_{attr}$ is the error metric of attribute a in the instance of the algorithm for α_i.

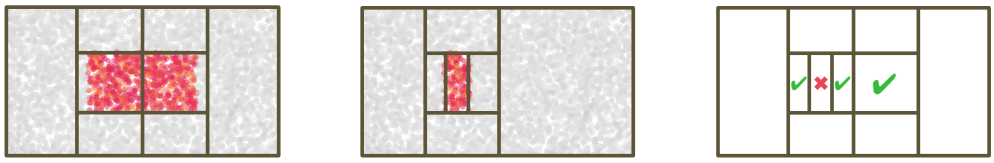
Synchronizing Outlier and Hold-out Partitioning

DT separately partitions outlier from hold-out input groups to avoid the complexity of computing the combined influence. It is tempting to compute the union of the input groups and execute the above recursive partitioner on the resulting set, however, it can result in over-partitioning. For example, consider α_2 and α_3 from Table 4-3. The outlier temperature readings (T6 and T9) are correlated with low voltage. If l_{α_2} and l_{α_3} are combined, then the error metric of the predicate $voltage < 2.4$ would still have high variance, and be falsely split further. In the worst case, the partitioner will create single-tuple partitions.

The result of the separate partitioning procedures are a separate set of partitions for the outliers ($partitions_O$) and the hold-outs ($partitions_H$). The final step is to combine them into a single partitioning, $partitions_C$. The goal is to distinguish partitions that influence

hold-out results from those that only influence outlier results. We do this by splitting partitions in $partitions_O$ along their intersections with partitions in $partitions_H$.

For example, $partitions_H$ in Figure 4-9 contains a partition that overlaps with two of the influential partitions in $partitions_O$. The splitting process distinguishes partitions that influence hold-out results (contains a red 'X') from those that only influence outlier results (contains a green check mark).



(a) $\text{partitions}_\text{O}$ (b) $\text{partitions}_\text{H}$ (c) $\text{partitions}_\text{C}$

Figure 4-9: Combined partitions of two simple outlier and hold-out partitionings

4.8.2 BOTTOM-UP (MC) PARTITIONER

The *MC* algorithm is a bottom-up approach for independent, anti-monotonic aggregates, such as $COUNT$ and SUM. It can be much more efficient than DT for these aggregates. The idea is to first search for influential single-attribute predicates, then intersect them to construct multi-attribute predicates. Our technique is similar to algorithms used for subspace clustering [5], so we will first sketch a classic subspace clustering algorithm, and then describe our modifications. The output is then sent to the *Merger*.

Subspace Clustering

The subspace clustering problem searches for all subspaces (hyper-rectangles) that are denser than a user defined threshold. The original algorithm, CLIQUE [5], and subsequent improvements, employs a bottom-up iterative approach that initially splits each continuous attribute into fixed size units, and every discrete attribute by the number of distinct attribute values. Each iteration computes the intersection of all units kept from the previous iteration whose dimensionality differ by exactly one attribute. Thus, the dimensionality of the units increase by one after each iteration. Non-dense units are pruned, and the remaining units are kept for the next iteration. The algorithm continues until no dense units are left. Finally, adjacent units with the same dimensionality are merged. The pruning step is possible because density (i.e. $COUNT$) is anti-monotonic because non-dense regions cannot contain dense sub-regions.

The intuition is to start with coarse-grained predicates (single dimensional), and improve the influence by adding additional dimensions that refine the predicates.

Algorithm 1 Pseudocode for the **MC** partitioning algorithm.

```
1:  function MC(O, H, V)
2:      predicates ← Null
3:      best ← Null
4:      while |predicates| > 0 do
5:          if predicates = Null then
6:              predicates ← initialize_predicates(O, H)
7:          else
8:              predicates ← intersect(predicates)
9:          best ← arg max_{p∈merged} inf(p)
10:         predicates ← prune(predicates, O, V, best)
11:         merged ← Merger(predicates)
12:         merged ← {p|p ∈ merged ∧ inf(p) > inf(best)}
13:         if merged.length = 0 then
14:             break
15:         predicates ← {p|∃_{p_m∈merged} p ≺_D p_m}
16:         best ← arg max_{p∈merged} inf(p)
17:     return best
18:
19: function PRUNE(predicates, O, V, best)
20:     ret = {p ∈ predicates|inf(O, ∅, p, V) < inf(best)}
21:     ret = {p ∈ ret| arg max_{t*∈p(O)} inf(t*) < inf(best)}
22:     return ret
```

Major Modifications

We have two major modifications to the subspace clustering algorithm. First, we merge adjacent units after each iteration to find the most influential predicate. If the merged predicate is not more influential than the optimal predicate so far, then the algorithm terminates.

Second, we modify the pruning procedure to account for two ways in which the influence metric is not anti-monotonic. The first case is when the user specifies a hold-out set. Consider the problem with a single outlier result, o, and a single hold-out result, h (Figure 4-10). A predicate, p, may be non-influential because it also influences a hold-out result (Figure 4-10.a), or because it doesn't influence the outlier result (Figure 4-10.b). In the former case, there may exist a predicate, $p' \prec_{l_o \cup l_h} p$ that only influences the outlier results. Pruning p would mistakenly also prune p'. In the latter case, p can be safely pruned. We distinguish

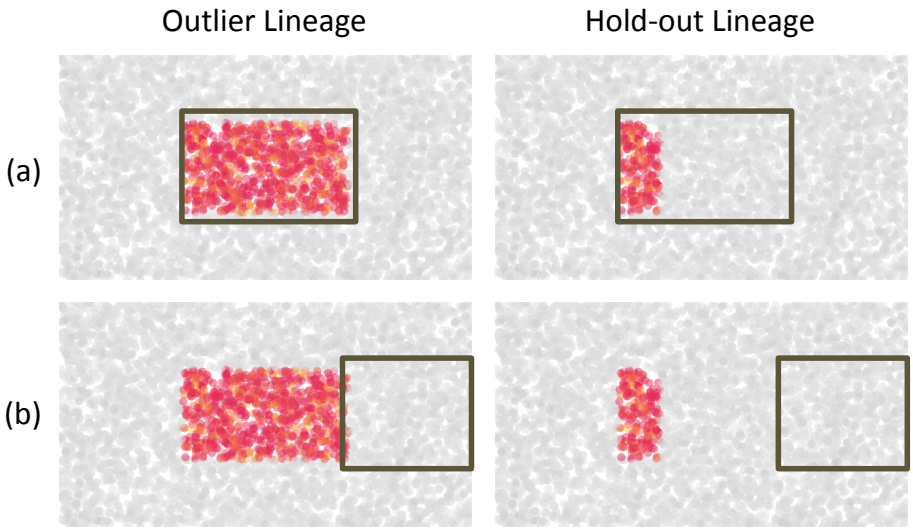


Figure 4-10: The predicates are not influential because they either (a) influence a hold-out result or (b) doesn't influence an outlier result.

these cases by pruning p based on its influence over only the outlier results, which is a conservative estimate of p's true influence.

The second case is because anti-monotonicity is defined for $\Delta(p)$, however influence is proportional to $\frac{\Delta(p)}{|p|^c}$, which is not anti-monotonic if $c > 0$. For example, consider three tuples with influences, $\{1, 50, 100\}$ and the operator SUM. The set's influence is $\frac{(1+50+100)}{3} = 50.3$, whereas the subset $\{50, 100\}$ has a higher influence of 75. It turns out that the anti-monotonicity property holds if, for a set of tuples T, the tuple with the maximum influence is less than the influence of T:

$$inf(t^*) < inf(T) \;\; |t^* = \arg\max_{t \in T} inf(t)$$

Algorithm 22 lists the pseudocode for the MC algorithm. The first iteration of the $WHILE$ loop initializes *predicates* to the set of single attribute predicates and subsequent iterations intersect all pairs in *predicates* (Lines 5-8). The best predicate so far, *best*, is updated, and then used to prune *predicates* (Lines 9,10). The resulting predicates are merged, and filtered for ones that are more influential than *best* (Lines 11-12). If none of the merged predicates are more influential than *best*, then the algorithm terminates. Otherwise *predicates* and *best* are updated, and the next iteration proceeds.

The pruning step first removes predicates whose influence, ignoring the hold-out sets, is less than the influence of *best*. It then removes those that don't contain a tuple whose individual influence is greater than *best*'s influence.

4.9 MERGER OPTIMIZATIONS

Section 4.6.3 describe a basic merging algorithm that scans the list of predicates and expands each one by repeatedly merging it with its adjacent predicates. It results in a list of merged predicates ordered by influence.

In this section, we propose several heuristic optimizations to the basic algorithm. In addition, we propose a second merging algorithm that can search for good predicates over a range of c hyperparameter values so that the merger is not limited to a single c value in each run. This is valuable when the user wants to try multiple c values to see how the top predicates change.

4.9.1 BASIC OPTIMIZATIONS

The main overheads in the basic merger are due to the cost of merging two predicates and applying the predicate to compute its influence, the number of predicates to expand, and the number of neighbors that are candidates for merging. This subsection presents optimizations that target the former two overheads when the aggregation function is independent.

Approximate Scorer

The first optimization seeks to completely avoid calling the *Scorer* when the operator is also incrementally removable (e.g., AVG, $STDDEV$). Instead, it uses state stored in existing predicates to apprimate the influence of the merged result.

Although the incrementally removable property already avoids recomputing the aggregate over the entire dataset, there is still the cost of evaluating the predicate on the input datasets. Doing this for every pair of neigboring predicates will still be very slow.

Recall that DT generates partitions where the tuples in a partition have similar influence. We modify DT to additionally record each partition's cardinality, and the tuple whose influence is closest to the mean influence of the partition. The *Merger* can use the aggregate's $state, update, remove$ and $recover$ functions to directly approximate the influence of a partition from the cached tuple.

Concretely, let partition p have cardinality N and its cached tuple be t. Let $m_t = state(t)$ and $m_D = state(D)$ be the states of $\{t\}$ and the dataset, then:

$$inf(p) \approx recover(remove(m_D, update(m_t, \cdots, m_t)))$$

where $update$ combines N copies of m_t. In other words, p's influence can be approximated by combining N copies of m_t, removing them from m_D, and calling $recover$.

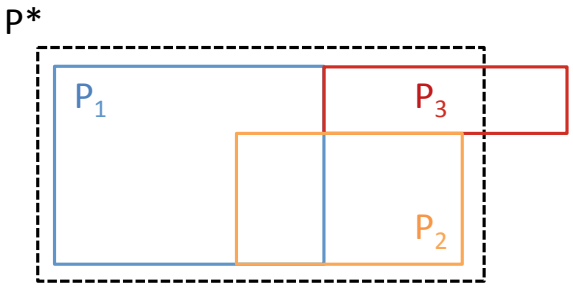


Figure 4-11: Merging partitions p_1 and p_2

Now consider merging partitions p_1 and p_2 into p^* as shown in Figure 4-11 and approximating its influence. This scenario is typically difficult because its not clear how the tuples in p_3 and $p_1 \cap p_2$ affect p^*'s influence. Similar to replicating the cached tuple multiple times to approximate a single partition, we estimate the number of cached tuples that p_1, p_2, and p_3 contribute.

We assume that tuples are distributed uniformly within the partitions. Let V_p and N_p be the volume and cardinality of partition p and let p_{ij} be a shorthand for $p_i \cap p_j$. Then the number of cached tuples from each partition n_p is computed as follows:

$$n_{p_1} = N_{p_1} \times \frac{V_{p_1} - 0.5V_{p_{12}}}{V_{p^*}}$$
$$n_{p_2} = N_{p_2} \times \frac{V_{p_2} - 0.5V_{p_{12}}}{V_{p^*}}$$
$$n_{p_3} = N_{p_3} \times \frac{V_{p_3 \cap p^*}}{V_{p^*}}$$

The *Merger* approximates a partition's influence from the input partitions by estimating the number of cached-tuples that each input partition contributes. Thus, the cost only depends on the number of intersecting partitions, rather than the size of the dataset.

We can prevent the approximation error for accumulating by periodically sending a merged predicate to the *Scorer* to compute its true influence, cardinality, and representative tuple.

Reducing Expandable Predicates

The second optimization reduces the number of predicates that need to be expanded by only expanding the predicates whose influences are within the top quartile. This is based on the intuition that the final predicate is most likely to influence predicates in the top

quartile, so it is inefficient to expand less influential predicates. This approach does not work for non-independent functions such as SUM because a predicate containing non-influential tuples may itself be influential. Section 4.7.2 illustrates an example.

4.9.2 SINGLE-PASS MERGING ALGORITHM

The previous algorithm finds the top predicates for an influence function that is parameterized with a fixed c value. Since the c value trades off the absolute amount of influence with the predicate's cardinality, it is desirable to find the best predicates for many different c values – ideally all values within a range. One possibility is to try different c values, however it is unclear which values to try because the best c value depends on human judgement. Thus, we might consider asking the user to manipulate c through an interface element and inspect the results. However, our user studies showed that the parameter leads to user confusion and is an ineffective design choice. In addition, each iteration requires running Scorpion again.

For these reasons, we have designed a single-pass merging algorithm to sweep through a range of c values to find the best predicates for each c value. The partitioning algorithms described in the previous section do not depend on c (with the exception of minor changes to MC), thus the primary challenge is designing a new merging algorithm to support this use case.

Preliminaries

The main insight is that a predicate p's influence can be represented by a curve parameterized by c. Recall that the influence function is computed (simplified to ignore λ and V) as:

$$inf_{agg}(O, H, p, c) = \underset{o \in O}{\text{avg}} \frac{\Delta_{agg}(o, p)}{|p(l_o)|^c} - \max_{h \in H} |\frac{\Delta_{agg}(p)}{1}| \tag{4.3}$$

Given a specific predicate and input dataset, the terms $\Delta_{agg}(o, p)$, $|p(l_o)|$, and the $\max_{h \in H}$ subexpression can be converted into constants k^o_Δ, k^o_{card}, k_H. This conversion allows us to simplify the equation to only depend on c:

$$inf_p(c) = \underset{o \in O}{\text{avg}} \frac{k^o_\Delta}{(k^o_{card})^c} - k_H \tag{4.4}$$

Since k^o_{card} is always positive, this function is monotonically decreasing (increasing) when the k^o_Δ values are positive (negative). Figure 4-12 illustrates the influence curves for two example predicates whose k^o_Δ values are positive. p_2 has the highest influence when $c \in [0, 0.15]$, whereas p_1 is optimal when $c > 0.15$. The grey dashed line depicts the frontier

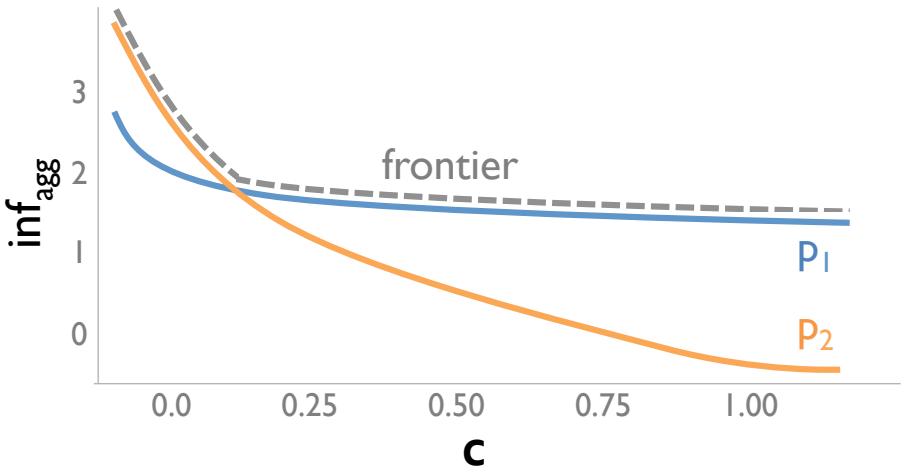


Figure 4-12: Influence curves for predicates p_1 and p_2, and the frontier (grey dashed line).

of the two predicates, as computed by the maximum influence over the set of predicates P:

$$inf_{frontier}(c, P) = \max_{p \in P} inf_p(c) \tag{4.5}$$

We say that a predicate p_1 *dominates* p_2 at c if $inf_{p_1}(c) \geq inf_{p_2}(c)$. A predicate p is called a *frontier predicate of* P if there exists a c such that p dominates the predicates in P:

$$\exists c \in [c_{min}, c_{max}] \; \forall p' \in P \; p(c) \text{ dominates } p'(c)$$

We also define the frontier of a set of predicates P as the subset of P that are frontier predicates:

$$frontier(P, c_{min}, c_{max}) = \{p \in P \mid \forall c \in [c_{min}, c_{max}] \wedge inf_{frontier}(c, P) = inf_p(c)\} \tag{4.6}$$

Thus the goal of the modified merging algorithm is to find the frontier predicates P within the predicate space $P_{A_{rest}}$ that maximizes the integral of its frontier within a user defined range of c values.

$$\arg\max_{P \in P_{A_{rest}}} \int_{c \in [c_{min}, c_{max}]} inf_{frontier}(c, P) \mathrm{d}c \tag{4.7}$$

Algorithm

Algorithm 18 lists the pseudocode for a greedy algorithm that approximates the solution for Equation 4.7. The algorithm tracks the current frontier predicates and iteratively merges the existing frontier predicates with their neighbors (line 4) until the frontier reaches a fixed point (line 6).

Algorithm 2 Pseudocode for single-pass merging algorithm.

```
1:  function FrontierMerger(P, c_min, c_max)
2:      while true do
3:          P_f ← FRONTIER(P, c_min, c_max)
4:          P' ← ⋃_{p∈frontier} expand(p)
5:          P'_f ← FRONTIER(P', c_min, c_max)
6:          if |P'_f − P_f| = 0 then
7:              return P_f
8:          P ← P'
9:
10: function FRONTIER(P, c_min, c_max)
11:     c_cur ← c_min
12:     p_cur ← arg max_{p∈P} inf_p(c_cur)
13:     frontier ← ∅
14:     while c_cur ≤ c_max do
15:         frontier ← frontier ∪ {p_cur}
16:         nextroots ← {(c_i, p) | p ∈ P ∧ c_i ∈ intersection(p_cur, p) ∧ c_i > c_cur}
17:         c_cur, p_cur ← arg min_{(c_i,p)∈nextroots} c_i
18:     return frontier
```

We compute $frontier(P, c_{min}, c_{max})$ by noting that a frontier predicate will continue to dominate until its curve intersects with that of another predicate. For example, Figure 4-12 illustrates that P_2 dominates P_1 at $c = 0$, and continues to dominate until it intersects with P_1 at $c = 0.15$. With this intuition, we developed an algorithm to compute the frontier in a single careful sweep of $c \in [c_{min}, c_{max}]$ by logging all of the intersection points where the dominating predicate changes. The algorithm initializes with the dominating predicate at $c = c_{min}$ (Lines 1-2). It repeatedly computes the intersection points (Line 14) between the current frontier predicate and each predicate in P, and picks the predicate with the closest intersection point (line 17) to replace the current frontier predicate.

Since there are no closed-form solutions to find the intersection points between the influence curves, we resort to numerical methods. This is expensive if we need to compute intersections between every pair of predicates. An alternative is to pre-compute each predicate's influence at N sample c values, and compute the dominating predicate at each sample.

As $N \to \infty$, the resulting frontier will converge with the solution in Algorithm 18. In practice, we find that $N \approx 50$ produces results that are comparable to that of the exact solution.

4.10 DIMENSIONALITY REDUCTION

Reducing the number of attributes in A_{rest} helps reduce the predicate space that Scorpion needs to consider, and is an optimization that can be applied independent of the particular partitioning and merging algorithm that is used.

One approach is to apply filter-based feature selection techniques [93] to the dataset. These techniques identify non-informative features by computing correlation or mutual information scores between pairs of attributes. For example, if we know that attributes *day* and *timestamp* are strongly correlated, then we can treat them as the same logical attribute e.g., *daytstamp*. A result predicate that contains the logical attribute, such as $daytstamp < July/01/2014\ 1PM$ can be expanded into $day < July/01/2014$ and $tstamp < July/01/2014\ 1PM$.

Attributes that are strongly correlated with A_{gb} are also unlikely to be of interest, and can be ignored. For example, if the query groups by *timestamp*, then predicates on *epoch* will simply select the same records as l_o and not provide any extra information.

In addition, the attributes could be ordered by importance, and Scorpion could preferentially split and merge attributes based on importance. This often makes sense when external information can help distinguish informative and actionable attributes (e.g., sensor) from non-actionable attributes (e.g., debug-level) or non-informative ones (e.g., epoch).

Scorpion currently supports ignoring attributes and relies on the client to specify attributes that can be ignored. We consider this decision as an orthogonal problem to the one in this chapter.

4.11 EXPERIMENTAL SETUP

The goal of these experiments is to gain an understanding of how the different partitioning and merging algorithms compare in terms of performance and answer quality. Furthermore, we want to understand how the c parameter impacts the types of predicates that the algorithms generate. We first use a synthetic dataset with varying dimensionality and task difficulty to analyze the algorithms, then anecdotally comment on the result qualities on 4 and 12 dimensional real-world datasets.

4.11.1 DATASETS

This subsection describes each dataset and their schema, attributes, query workload, and properties of the outlier tuples.

Synthetic Dataset (**SYNTH**)

The synthetic dataset is used to generate ground truth data to compare our various algorithms. We use a simple group-by SQL query template and use SUM or AVG as the aggregation function to match the MC and DT algorithms:

$$\text{SELECT } \mathrm{A_d},\ \mathrm{agg(A_v)} \text{ FROM synthetic GROUP BY } \mathrm{A_d} \tag{Q3}$$

The data consists of a single group-by attribute A_d, one value attribute A_v that is used to compute the aggregate result, and n dimension attributes $A_1, \cdots, A_n$ that are used to generate the explanatory predicates. The value and dimension attributes have a domain of $[0, 100]$. We generate 10 distinct A_d values (to create 10 groups), and each group contains 2,000 tuples randomly distributed in the n dimensions. The A_v values are drawn from one of three gaussian distributions, depending on if the tuple is a normal or outlier tuple, and the type of outlier. Normal tuples are drawn from $\mathcal{N}(10, 10)$. To illustrate the effects of the c parameter we generate high-valued outliers, drawn from $\mathcal{N}(\mu, 10)$, and medium valued outliers, drawn from $\mathcal{N}(\frac{\mu+10}{2}, 10)$. $\mu > 10$ is a parameter to vary the difficulty of distinguishing normal and outlier tuples. The problem is harder the closer μ is to 10. The hold-out groups exclusively sample from the normal distribution, while the outlier groups sample from all three distributions.

We generate the outlier groups by creating two random n dimensional hyper-cubes over the n attributes where one is nested inside the other. The *outer cube* samples from the median distribution and the *inner cube* samples from the high valued distribution. Each cube contains *perc*% of the volume of its immediate enclosing cube. Since points are distributed uniformly, each cube also contains *perc*% of the tuples in its enclosing cube. The tuples outside of the outer cube are normal.

For example, Figure 4-13 illustrates an example 2D dataset and query results. The top graph renders the aggregate results the that a user would see, and bottom shows input tuples of one outlier result and one hold-out result. The right scatterplot visualizes the tuples in an outlier group with $\mu = 90$ and $perc = 25$. The outer cube (orange points) encloses $A_1 \in [42, 92], A_2 \in [37, 87]$ and the inner cube (red points) encloses $A_1 \in [52, 77], A_2 \in [44, 69]$.

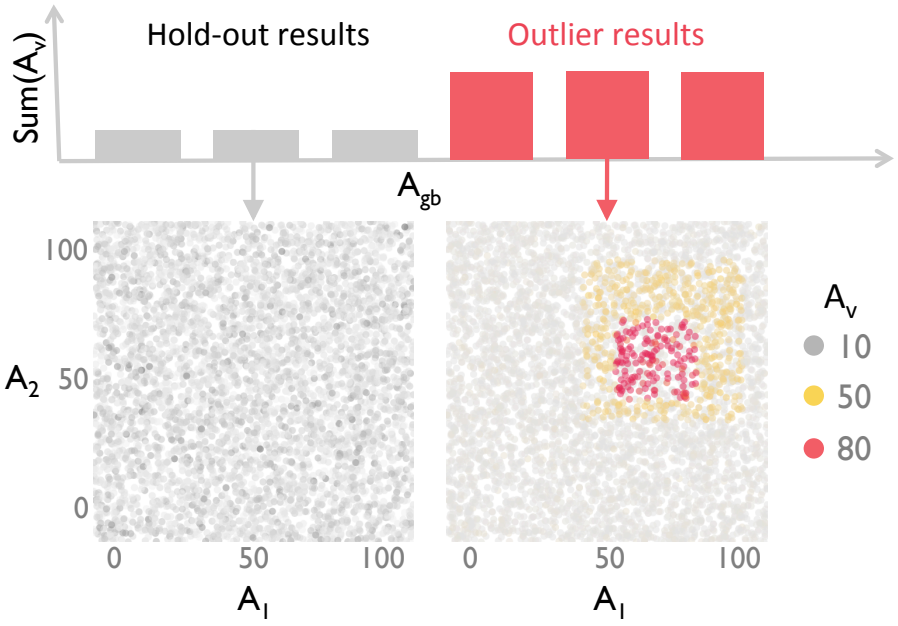


Figure 4-13: Visualization of outlier and hold-out results and tuples in their input groups from a 2-D synthetic dataset. The colors represent normal tuples (light grey), medium valued outliers (orange), and high valued outliers (red).

In the experiments, we flag the 5 outlier aggregate results, and use the other 5 as hold-outs. We also vary the dimensionality from 2 to 4, and the difficulty between *Easy* ($\mu = 80$) and *Hard* ($\mu = 30$), For example, *SYNTH-2D-Easy* describes a 2-dimensional dataset where $\mu = 80$.

Intel Dataset (**INTEL**)

The Intel dataset contains 2.3 million rows, and 6 attributes. Four of the attributes, *sensorid*, *humidity*, *light*, and *voltage* are used to construct explanations. All of the attributes are continuous, except for *sensorid*, which contains ids of the 61 sensors.

We use two queries for this experiment, both related to the impact of sensor failures on the standard deviation of the temperature. The following is the general query template, and contains an independent aggregate:

SELECT truncate('hour', time) as hour, STDDEV(temp) (Q4)
FROM readings
WHERE *STARTDATE* $\leq$ time $\leq$ *ENDDATE* GROUP BY hour

The first query occurs when a single sensor ($sensorid = 15$) starts dying and generating temperatures above $100^o c$. The user selects 20 outliers and 13 hold-out results, and specifies that the outliers are too high.

The second query is when a sensor starts to lose battery power, indicated by low voltage readings, which causes above $100^o c$ temperature readings. The user selects 138 outliers and 21 hold-out results, and indicates that the outliers are too high.

Campaign Dataset (**EXPENSE**)

The expenses dataset [9] contains all campaign expenses between January 2011 and July 2012 from the 2012 US Presidential Election. The dataset contains 116448 rows and 14 attributes (e.g., recipient name, dollar amount, state, zip code, organization type), of which 12 are used to create explanations. The attributes are nearly all discrete, and vary in cardinality from 2 to 18 thousand (recipient names). Two of the attributes contain 100 distinct values, and another contains 2000.

The SQL query uses an independent, anti-monotonic aggregate and sums the total expenses per day in the Obama campaign. It shows that although the typical spending is around $5,000 per day, campaign spent up to $13 million per day on media-related purchases (TV ads) in June.

$$\begin{aligned} &\text{SELECT sum(disb\_amt)} \\ &\text{FROM expenses WHERE candidate } = \text{'Obama'} \\ &\text{GROUP BY date} \end{aligned} \tag{Q5}$$

We flag 7 outlier days where the expenditures are over $10M, and 27 hold-out results from typical days.

4.11.2 METHODOLOGY

Our experiments compare Scorpion using the three partitioning algorithms along metrics of precision, recall, F-score and runtime. We compute precision and recall of a predicate, p, by comparing the set of tuples in $p(g_O)$ to a ground truth set. The F-score is defined as the harmonic mean of the precision and recall:

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

[9] http://www.fec.gov/disclosurep/PDownload.do

The *NAIVE* algorithm described in Section 4.6.2 is clearly exponential and is unacceptably slow for any non-trivial dataset. We modified the exhaustive algorithm to generate predicates in order of increasing complexity, where complexity is terms of the number and size of values in a discrete clause, and the number of clauses in the predicate. The modified algorithm uses two outer loops that increases the maximum allowed complexity of the discrete clauses and the maximum number of attributes in a predicate, respectively, and an inner loop that iterates through all combinations of attributes and their clauses. When the algorithm has executed for a user specified period of time, it terminates and returns the most influential predicate generated so far. In our experiments, we ran the exhaustive algorithm for up to 40 minutes, and also logged the best predicate found so far every 10 seconds.

The current Scorpion prototype is implemented in Python 2.7 as part of an end-to-end data exploration tool. Relations are encoded as tables in the Orange [34] machine learning package, and predicates are evaluated as full table scans. Scorpion can be installed using the following commands:

```
pip install scorpion  # installs Scorpion
pip install dbwipes   # installs visualization frontend
```

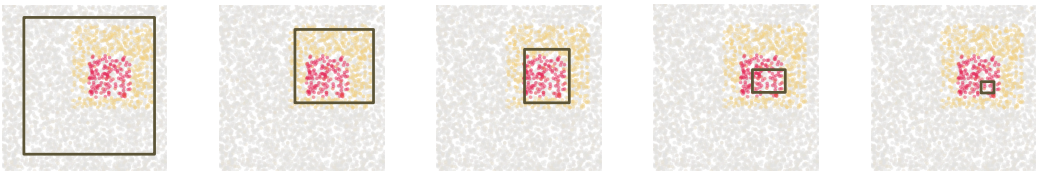
The experiments are run on a Macbook Pro (OS-X Lion, 8GB RAM). The influence scoring function was configured with $\lambda = 0.5$. The *Naive* and *MC* partitioner algorithms were configured to split each continuous attribute's domain into 15 equi-sized ranges. The *DT* algorithm was configured with $tau_{min} = 0.001$, $tau_{max} = 0.05$, and $\epsilon = 0.1\%$,

4.12 SYNTHETIC DATASET EXPERIMENTS

Our first set of experiments use the 2D synthetic datasets to highlight how the c parameter impacts the quality of the optimal predicate. We execute the *NAIVE* algorithm until completion and show how the predicates and accuracy statistics vary with different c values. The second set of experiments compare the *DT*, *MC* and *NAIVE* algorithms by varying the dimensionality of the dataset and the c parameter. The final experiment introduces a caching based optimization for the *DT* algorithm and the *Merger*.

4.12.1 NAIVE ALGORITHM

Figure 4-14 plots the optimal predicate that *Naive* finds for different c values on the SYNTH-2D-Hard dataset. When $c = 0$, the predicate encloses all of the outer cube, at the expense of including many normal points. When $c = 0.05$, the predicate contains most of the outer



(a) c = 0 (b) c = 0.05 (c) c = 0.1 (d) c = 0.2 (e) c = 0.5

Figure 4-14: Optimal NAIVE predicates for SYNTH-2D-Hard

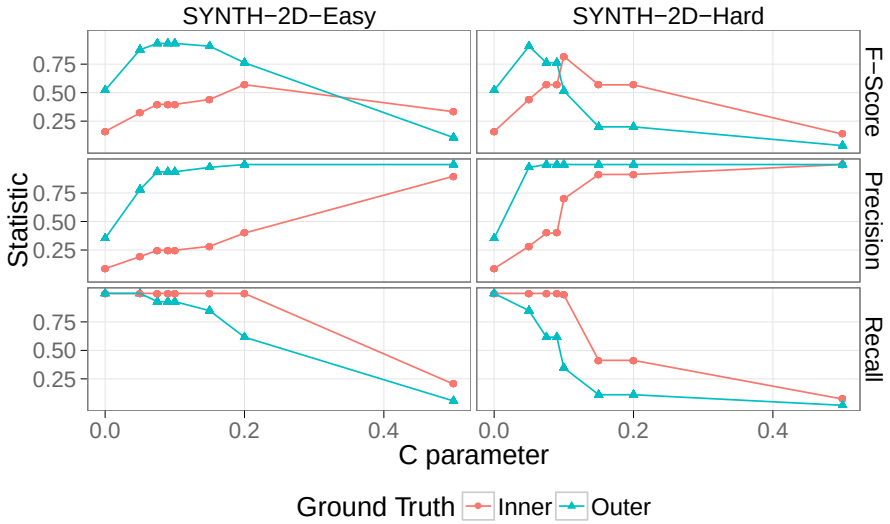


Figure 4-15: Accuracy statistics of NAIVE as c varies using two sets of ground truth data.

cube, but avoids regions that also contain normal points. Increasing c further reduces the predicate and exclusively selects portions of the inner cube.

It is important to note that all of these predicates are correct and influence the outlier results to a different degree because of the c parameter. This highlights the fact that a single best predicate is ill-defined because the actual ground truth depends on the user. For this reason, we simply use the tuples in the inner and outer cubes of the synthetic datasets as surrogates for two possible versions of ground truth.

Figure 4-15 plots the accuracy statistics as c increases. Each column of figures plots the results of a dataset, and each curve uses the outer or inner cube as the ground truth when computing the accuracy statistics. Note that for each dataset, the points for the same c value represent the same predicate. As expected, the F-score of the outer curve peaks at a lower c value than the inner curve. This is because the precision of the outer curve quickly approaches 1.0, and further increasing c simply reduces the recall. In contrast, the recall of the inner curve is maximized at lower values of c and reduces at a slower pace. The precision

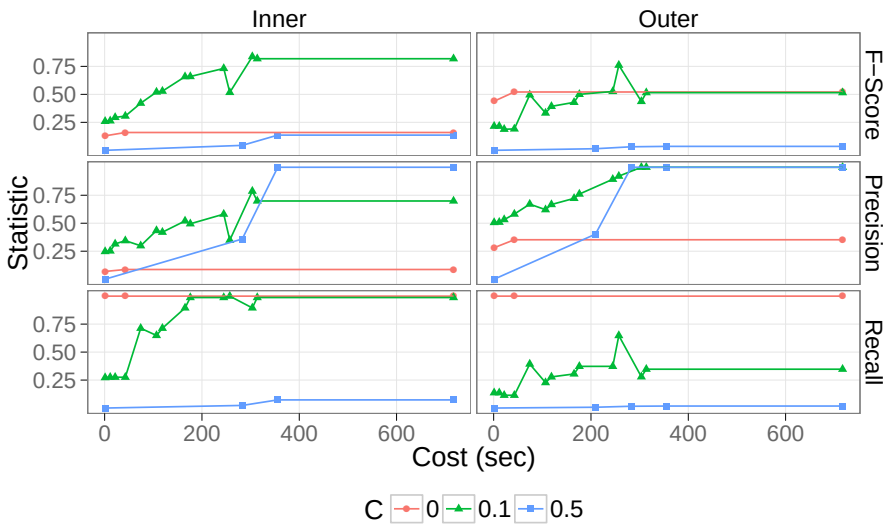


Figure 4-16: Accuracy statistics as execution time increases for NAIVE on SYNTH-2D-Hard

statistics of the inner curve on the Easy dataset increases at a slower rate because the value of the outliers are much higher than the normal tuples, which increases the predicate's Δ values.

Figure 4-16 depicts the amount of time it takes for Naive to converge when executing on SYNTH-2D-Hard. The left column computes the accuracy statistics using the inner cube as ground truth, and the right column uses the outer cube. Each point plots the accuracy score of the most influential predicate so far, and each curve is for a different c value. *NAIVE* tends to converge faster when c is close to zero, because the optimal predicate involves fewer attributes. The curves are not monotonically increasing because the the optimal predicate as computed by influence does not perfectly correlate with the ground truth that we selected.

Takeaway: Although the F-score is a good proxy for result quality, it can be artificially low depending on the value of c. NAIVE converges (relatively) quickly when c is very low but it can be very slow at high c values.

4.12.2 COMPARING ALGORITHMS

The following experiments compare the accuracy and runtime of the *DT*, *MC* and *NAIVE* algorithms. Figure 4-17 varies the c parameter and computes the accuracy statistics using the outer cube as the ground truth. Both *DT* and *MC* generate results comparable with those from the *NAIVE* algorithm. In particular, the maximum F-scores are similar.

Figure 4-18 compares the F-scores of the algorithms as the dimensionality varies from

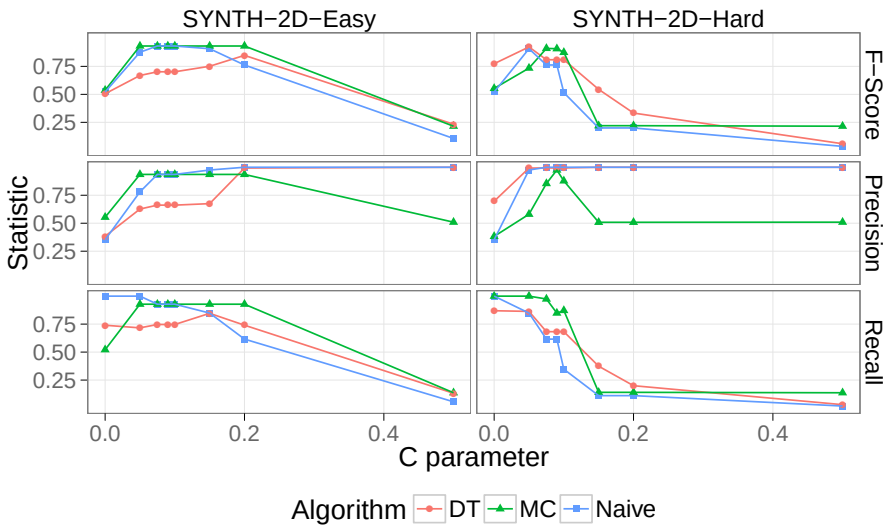


Figure 4-17: Accuracy measures as c varies

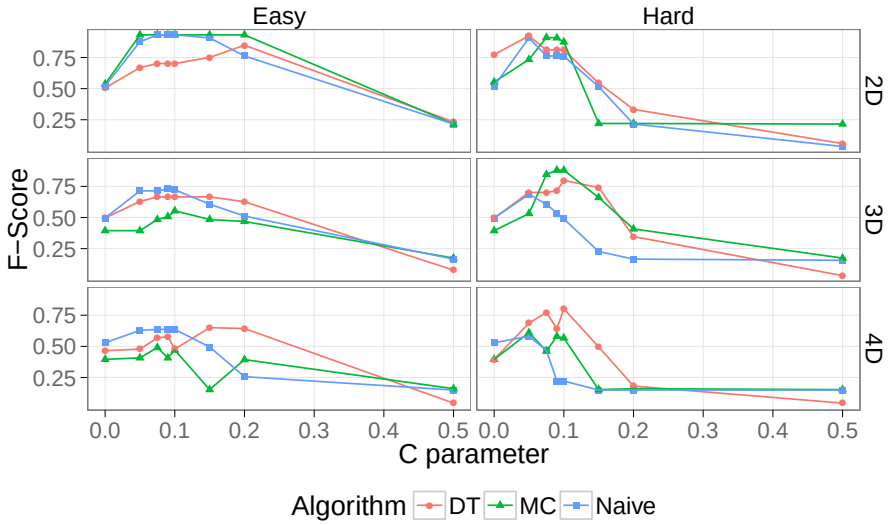


Figure 4-18: F-score as dimensionality of dataset increases

2 to 4. Each row and column of plots corresponds to the dimensionality and difficulty of the dataset, respectively. As the dimensionality increases, *DT* and *MC* remain competitive with *NAIVE*. In fact, in some cases *DT* produces better results than *NAIVE*. Partly because because *NAIVE* splits each attribute into a pre-defined number of intervals, whereas *DT* can split the predicates into any granularity, and partly because *NAIVE* doesn't terminate within the 40 minutes at higher dimensions – running it to completion would generate the optimal predicate.

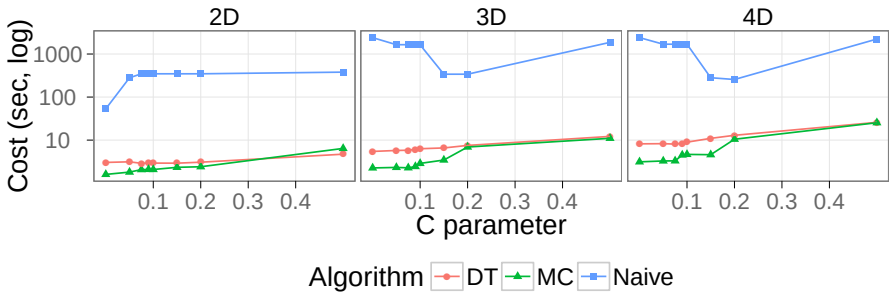


Figure 4-19: Cost as dimensionality of Easy dataset increases

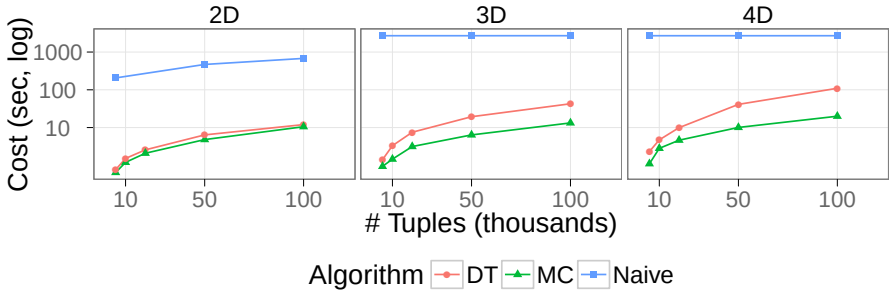


Figure 4-20: Cost as size of Easy dataset increases (c=0.1)

Figure 4-19 compares the algorithm runtimes while varying the dimensionality of the Easy synthetic datasets. The *NAIVE* curve reports the earliest time that *NAIVE* converges on the predicate returned when the algorithm terminates. We can see that *DT* and *MC* are up to two orders of magnitude faster than Naive. We can also see how *MC*'s runtime increases as c increases because there are less opportunities to prune candidate predicates.

Figure 4-20 uses the Easy datasets and varies the number of tuples per group from 500 (5k total tuples) to 10k (100k total tuples) for a fixed $c = 0.1$. The runtime is linear with the dataset size, but the slope increases super-linearly with the dimensionality because the number of possible splits and and merges increases similarly. We found that *DT* spends significant time splitting non-influential partitions because the standard deviation of the tuple samples are too high. When we re-ran the experiment by reducing the variability by drawing normal tuples from $\mathcal{N}(10, 0)$ reduces the runtime by up to 2×. We leave more advanced optimization techniques, e.g., early pruning, parallelism to future work.

Takeaway: DT and MC generate results competitive with the exhaustive NAIVE algorithm and reduces runtime costs by up to 150×. Algorithm performance relies on data properties, and scales exponentially with the dimensionality in the worst case. DT's results may have

higher F-scores than NAIVE because it can progressively refine the predicate granularity.

4.12.3 CACHING OPTIMIZATION

The previous experiments showed that the result predicates are sensitive to c, thus the user or system may want to try different values of c (e.g., via a slider in the UI or automatically). *DT* can cache and re-use its results because the partitioning algorithm is agnostic to the c parameter. Thus, the *DT* partitioner only needs to execute once for Scorpion queries that only change c.

The *Merger* can similarly cache its previous results because it executes iteratively in a deterministic fashion – increasing the c parameter simply reduces the number of iterations that are executed. Thus Scorpion can initialize the merging process to the results of any prior execution with a higher c value. For example, if the user first ran a Scorpion query with $c = 1$, then those results can be re-used when the user reduces c to 0.5.

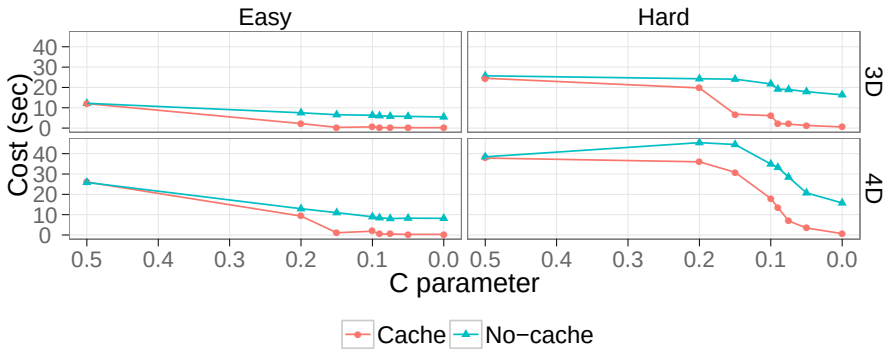


Figure 4-21: Cost with and without caching enabled

Figure 4-21 executes Scorpion using the *DT* partitioner on the synthetic datasets. We execute on each dataset with decreasing values of c (from 0.5 to 0), and cache the results so that each execution can benefit from the previous one. Each sub-figure compares Scorpion with and without caching. It is most beneficial to cache *Merger* results at lower c values because more predicates are merged so there are less predicates to consider merging. When c is high, most predicates are not expanded, so the cache doesn't reduce the amount of work that needs to be done.

Takeaway: Caching DT and Merger results for low c values reduces execution cost by up to 25×.

4.13 REAL-WORLD DATASETS

To understand how Scorpion performs on real-world datasets, we applied Scorpion to the INTEL and EXPENSES workloads. Since there is no ground truth, we present the predicates that are generated and comment on the predicate quality with respect to our expectations and further analyses. The algorithms all completed within a few seconds, so we focus on result quality rather than runtime. In each of the workloads, we vary c from 1 to 0, and record the resulting predicates.

4.13.1 INTEL DATASET

For the first workload, the outliers are generated by Sensor 15, so Scorpion consistently returns:

$$\text{sensorid} = 15$$

However, when c approaches 1, Scorpion generates the predicate:

$$\text{light} \in [0, 923] \ \& \ \text{voltage} \in [2.307, 2.33] \ \& \ \text{sensorid} = 15$$

It turns out that although Sensor 15 generates all of the high temperature readings, the temperatures vary between are $20^o c$ higher when its voltage, and surprisingly, light readings are lower.

In the second workload, Sensor 18 generates the anomalous readings. When $c = 1$, Scorpion returned:

$$\text{light} \in [283, 354] \ \& \ \text{sensorid} = 18$$

Sensor 18's voltage is abnormally low, which causes it to generate high temperature readings ($90^o c - 122^o c$). The readings are particularly high ($122^o c$) when the light levels are between 283 and 354. At lower c values, Scorpion returns:

$$\text{sensorid} = 18.0$$

In both workloads, Scorpion identified the problematic sensors and distingushed between extreme and normal outlier readings.

4.13.2 CAMPAIGN EXPENSES DATASET

In this workload, we defined the ground truth as all tuples where the expense was greater than \$1.5M. The aggregate was SUM and all of the expenses were positive so we executed the MC algorithm. When $c \in [0.2, 1]$, Scorpion generated the predicate:

$$\text{recipient\_st} = \text{`DC}' \ \&\ \text{recipient\_nm} = \text{`GMMB INC.}' \ \&$$
$$\text{file\_num} = 800316 \ \&\ \text{disb\_desc} = \text{`MEDIA BUY}'$$

Although the F-score is 0.6 due to low recall, this predicate best describes Obama's highest expenses. The campaign submitted two "GMMB INC." related expense reports. The report with file_num = 800316 spend an average of \$2.7M. When $c \leq 0.1$, the file_num clause is removed, and the predicate matches all $\$1 + M$ expenditures for an average expenditure of \$2.6M.

4.14 CONCLUSION

As data becomes increasingly accessible, data analysis capabilities will shift from specialists into the hands of end-users. These users not only want to navigate and explore their data, but also probe and understand why outliers in their datasets exist. Scorpion helps users understand the origins of outliers in aggregate results computed over their data. In particular, we generate human-readable predicates to help *explain* outlier aggregate groups based on the attributes of tuples that contribute to the value of those groups, and introduced a notion influence for computing the effect of a tuple on an output value. Identifying tuples of maximum influence is difficult because the influence of a given tuple depends on the other tuples in the group, and so a naive algorithm requires iterating through all possible inputs to identify the set of tuples of maximum influence. We then described three aggregate operator properties that can be leveraged to develop efficient algorithms that construct influential predicates of nearly equal quality to the exhaustive algorithm using orders of magnitude less time. Our experiments on two real-world datasets show promising results, accurately finding predicates that "explain" the source of outliers in a sensor networking and campaign finance data set.

5 Exploratory & Explanatory Visualization

The previous chapters laid the foundations for an explanatory visualization system. Chapter 3 described a provenance managament system that can efficiently track fine-grained record-level provenance and Chapter 4 developed the algorithms to use this provenance information to generate hypotheses that explain anomalies in aggregation query results. The missing piece is the interface for using these results as part of visual data analysis.

This chapter introduces DBWipes, an end-to-end visual analytics system that brings together the functionalities introduced in the previous chapters. Users can point DBWipes at a database and generate visualizations for aggregation queries, and interactively filter and navigate through the dataset. The system is integrated with Scorpion, so users can ask questions about anomalies in the visualization and assess and compare the quality of the generated explanations. We first introduce the basic DBWipes interface for querying and navigation, then describe the interface for interacting with Scorpion, and finally present the results of a user study to assess the efficacy of Scorpion's interface for analyzing visualization outliers.

5.1 BASIC DBWIPES INTERFACE

DBWipes is designed to facilitate rapid navigation though a dataset in the spirit of systems such as Splunk [121] and Tableau [102]. Similar to these systems, DBWipes renders a primary visualization and provides an faceted navigation interface to interactively specify filters over the dataset. In contrast to these systems, DBWipes also provides features that help assess how much subsets of the data impact outliers in the visualization.

The goal of the DBWipes system is to help users see an overview of the dataset, filter the dataset by combinations of attribute values, evaluate the impact of explanations in the form of predicates, and visualize aggregated statistics. To this end, we developed three interface components (shown in Figure 5-1) to address these goals. The left-hand column (A) shows the *faceting interface*, which renders an overview of each dataset attribute as a value

distribution and provides controls for users to interactively filter the dataset by forming conjunctive predicates. The *contextual panel* in the center column (B) lists different classes of filters that have been applied to the query. The right-hand column (C) contains the *main visualization*, which compares the results of the aggregation query over different subsets of the data. In this section, we describe these main components in more detail.

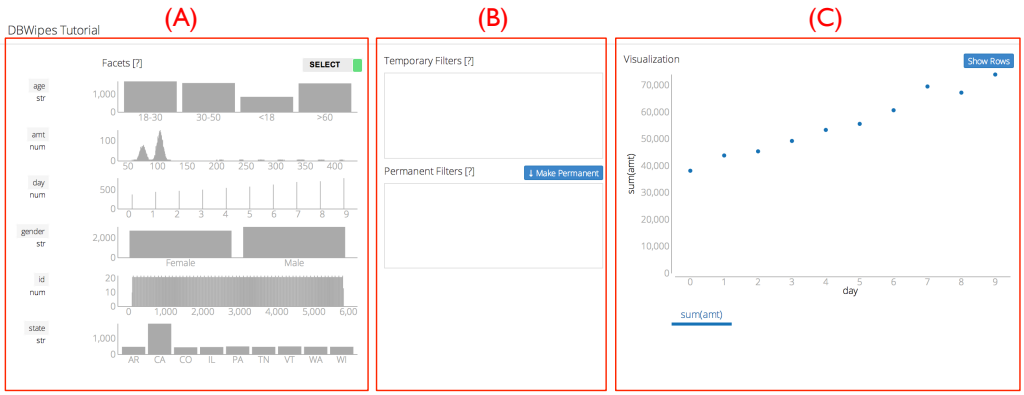


Figure 5-1: Basic DBWipes interface.

5.1.1 FACETING INTERFACE

The faceting interface (Figure 5-1(A)) provides faceted navigation between attribute distributions and the main visualization. The attributes in the database are rendered as rows in the interface; the left column lists the attribute name and type, and the right column renders a distribution of the attribute values as a bar chart. The attributes are listed in the same order as they appear in the table's schema definition, however alternative ordering (e.g., by statistics over the attribute values) are possible as well.

DBWipes currently renders univariate distributions where the x-axis lists each attribute value (or value range if the attribute type is quantatative) and the y-axis represents the number of database records with the corresponding attribute value(s). The y-axes can be rendered in log-scale if the variance in the cardinalities is significant. For example, the distribution for the *State* attribute shows that there are significantly more sales records in California than the other states.

Specifying Filters

Users select ranges in the attribute distributions (also called brushing) to specify conjunctive predicates over the dataset. We call these predicates *facet selections*. For example, Figure 5-2 shows the result of brushing the *Female* value of the gender attribute (B), which specifies the

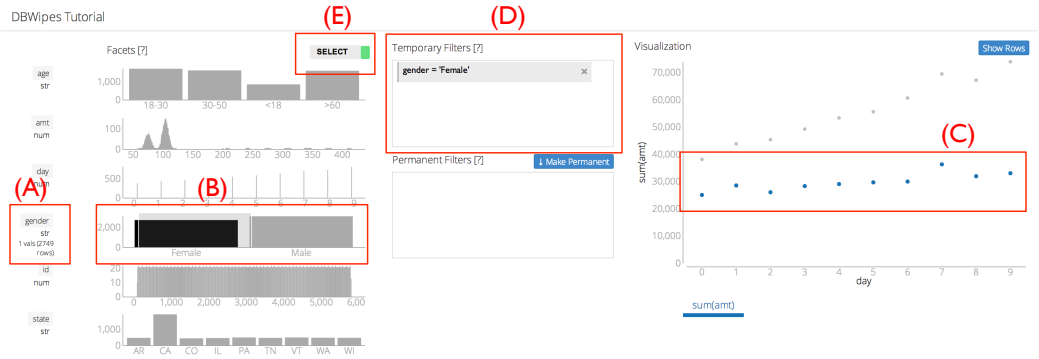


Figure 5-2: Faceted navigation using DBWipes.

predicate $gender = Female$. The bars corresponding to the selected values are highlighted in black, and statistics about the selected values (the number of distinct selected values and the number of records that match the per-attribute predicate) are listed in the left column under the attribute value (A). Handles on the selection can be used to interactively move and resize the selection, and clicking outside of the selection clears it.

Brushing multiple attributes result specifies the conjunction of the individual attribute predicates. DBWipes does not currently support disjunctions, due to the risk of complicating the interface.

Specifying facet selections will temporarily update the interface to reflect the query results over the filtered data. The update is temporary because interacting with the facet selections changes the predicate. The *main visualization* turns the original visualization grey and overlays the updated query results in color so that users can easily compare the predicate's effects (C). In addition, the facet selection is listed textually in the middle column's *Temporary Filters* section. Clicking the "×" button in the textual representation removes the predicate and clears the corresponding selection in the faceting interface.

Toggling Negation

The user can negate the predicate listed in the temporary filter by toggling the select/remove switch (Figure 5-2(E).) This is a proxy for the amount that the predicate contributes to the query's result values. For example, Figure 5-3 shows the results of toggling the switch in the example interface. The temporary filter has been negated to $not(gender = Female)$ (B), and the main visualization is updated to reflect the negated predicate (C). The result shows that ignoring female sales uniformly shifts the distribution down, but does not affect the slope of the distribution. This suggests that female sales may not be the primary contributor to the upward trend.

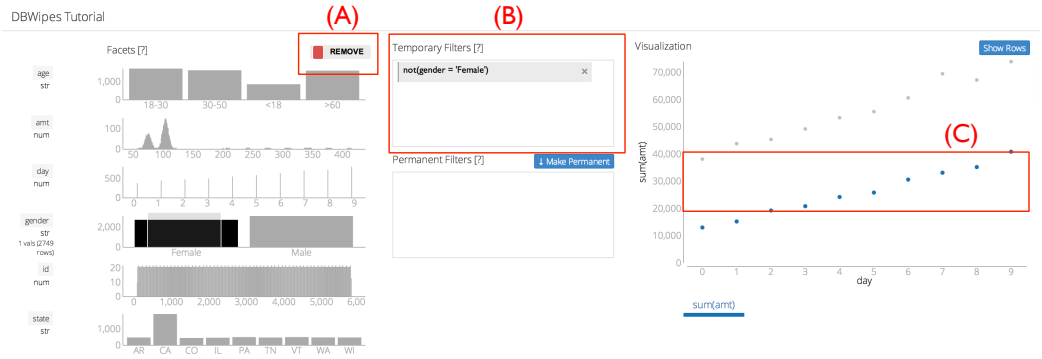


Figure 5-3: Negating a predicate illustrates its contributions to the aggregated results.

If the aggregation operator is *sum*, it is possible to directly infer this result from the non-negated predicate by mentally subtracting the updated values from the orginials. However, this feature is important for aggregation operators such as *average* or *standard deviation*, where estimating the amount of contribution is non-trival or even impossible. Our experiments in Section 5.9 found that this is indeed the case.

Permanent Filters

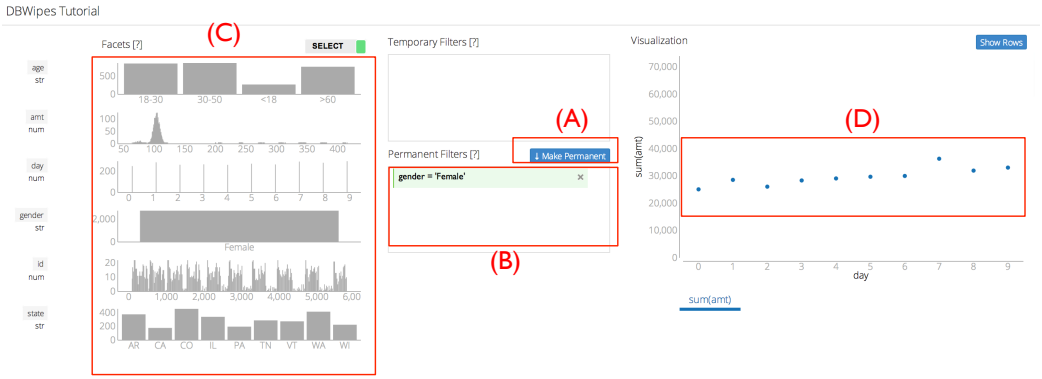


Figure 5-4: Setting a predicate as a permanent filter.

When the user specifies a predicate as a *Permanent Filter*, it has the effect of re-initializing the DBWipes interface with an updated aggregation query containing the predicate (Figure 5-4). This will naturally update the main visualization (D) as well as the distributions in the faceting interface (C). Users click on the "Make Permanent" button to add the current temporary filters to the list of permanent filters. We distinguish between permanent and temporary filters because updating the distributions in the faceting interface requires computing an aggregation query for each attribute in the table. This can be very expensive

for tables with many attributes (some datasets contain almost 2000 attributes).

5.2 SCORPION INTERFACE

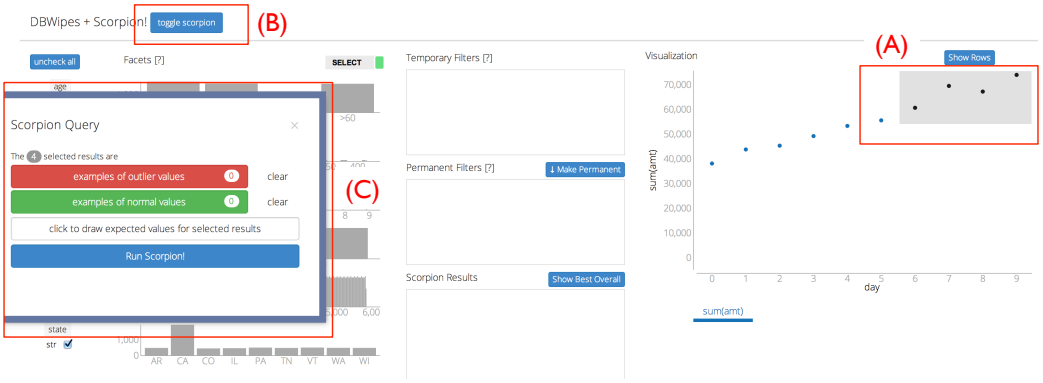


Figure 5-5: Scorpion query form interface.

Scorpion extends the DBWipes interface by allowing users to select anomalies in the main visualization and ask questions about them (Figure 5-5). The user can bring up the Scorpion interface (C) by selecting a set of points in the main visualization (A) or clicking "Toggle Scorpion" (B). The form contains two buttons to specify the user's selection as examples of outlier values or as normal values. A badge within each button shows the total number of outliers and normal results that have been specified. Scorpion compares the mean values of the outlier and normal examples to decide if the outlier values are too low or too high.

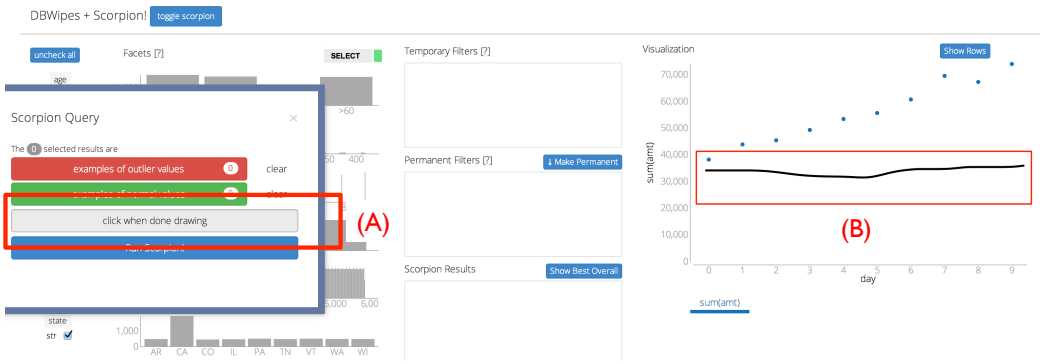


Figure 5-6: Interface to manually specify an expected trend.

Alternatively, the user can explicitly specify the desired value of each outlier value by clicking on the "Click to draw expected values for selected results" button (Figure 5-6(A)) and

drawing a desired trend line (B). DBWipes will compute the expected values by interpolating along the drawn line.

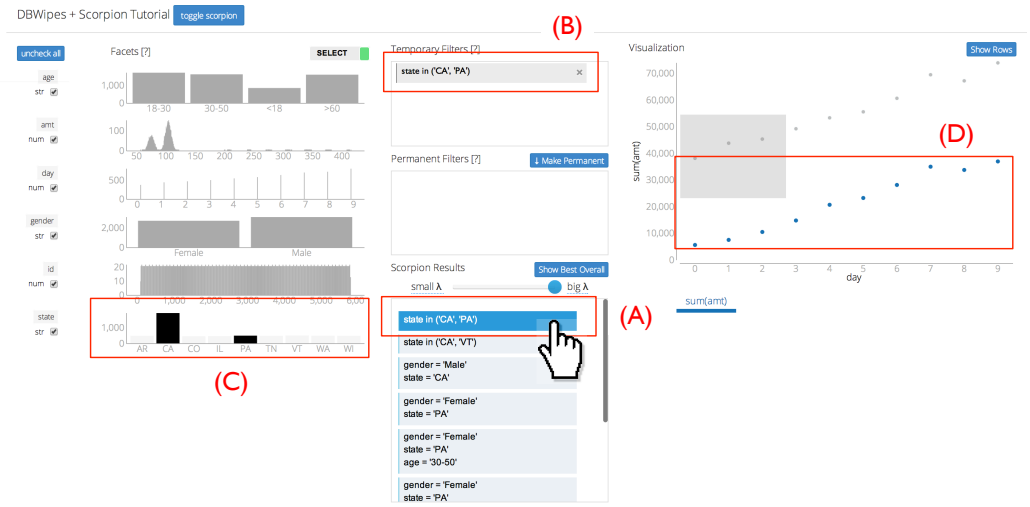
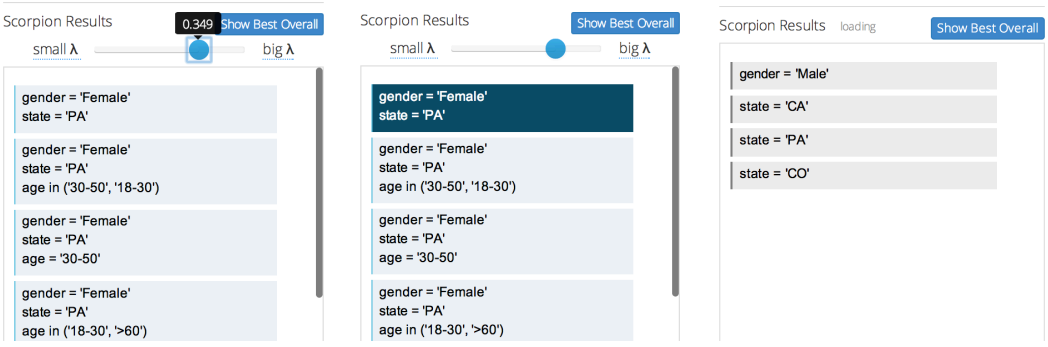


Figure 5-7: Selecting a Scorpion result in DBWipes.

Scorpion generates explanations as a list of predicates shown in the *Scorpion Results* section at the bottom of the center contextual panel. By default, DBWipes lists the top results for every c parameter between 0.1 and 1 (see Chapter 4 Section 4.4.3 for a description of the c parameter). The results are listed from the largest absolute impact on the outliers (low c parameter) to the largest impact per record (high c parameter.) The layout is intended to be consistent with that of the *Temporary* and *Permanent Filters*.



(a) The λ-slider trades off high absolute impact with high per-record impact.

(b) A locked result is rendered using a dark navy fill.

(c) The interface updates a list of the top results Scorpion has found so far.

DBWipes adds a slider so users can specify different values of the parameter λ[1] and view the top results for selected parameter value. For example, Figure 5-8a depicts the

[1]Although it is admittedly confusing, the DBWipes interface calls Scorpion's c parameter λ because λ is

top predicates for $\lambda = 0.349$, which are dominated by subsets of the predicate $gender = Female$ & $state = PA$.

Users can hover over a result to view its effects on the aggregated query (Figure 5-7). The result turns bright blue (A), and it is added as a *temporary filter* (B). The corresponding attribute values in the faceting interface are highlighted (C), and the main visualization updates to reflect the temporary filter (D). In this case we find that the predicate $state \in \{CA, PA\}$ matches records with a strong upward trend similar to the trend in the complete dataset.

When the cursor moves away from a result, the interface automatically reverts to the orginial query. If the user moves the cursor between two Scorpion results in order to compare their effects in the visualization, this visualization will swap between the first result, the original query, and the second result. This intermediate visualization state makes it difficult to directly compare the two results. To avoid this issue, users can *lock* a result in place by clicking on it. This colors the result as dark navy (Figure 5-8b) and ensures that the interface reverts to the locked result rather than the original query whenever the cursor moves away from any result. Now, the interface will continue to show $gender = Female$ & $state = PA$ until the user hovers over another result.

While Scorpion is running, DBWipes updates the interface with the best results that have been found so far (Figure 5-8c). These partial results are rendered in grey to distinguish them from Scorpion's final results. Users can select and lock the results in a consistent manner as with the final results. The main distinction is the absence of the λ slider, which is only shown for the final results.

5.3 IMPLEMENTATION

The DBWipes prototype is implemented as a HTML and ECMAScript browser application hosted from a Python server that communicates with a PostgreSQL backend. The browser application translates user interactions into SQL queries sent to the backend, which executes querys, caches intermediate results, and interfaces with Scorpion. We currently support aggregation queries with a single group-by attribute and over a single table. However DBWipes supports multiple aggregation statements in the SELECT clause, and renders each statement as a separate series with different colors in the visualization.

DBWipes is integrated as the visual analytics system for DataHub [15], a data hosting platform developed at MIT, University of Maryland and UIUC, that provides functionality

more commonly recognized as a system parameter. Thus, we use λ to refer to Scorpion's c parameter in the rest of this chapter.

to upload, clean, version and share datasets. Users can use DataHub to upload their datasets, and interact with them using DBWipes.

5.4 EXPERIMENTAL SETUP

We conducted a comparative user study between DBWipes with and without the Scorpion interface. Users performed three analysis tasks to explain outliers in a visualization and our goal was to compare the task completion times, the usefulness of the Scorpion interface, and understand different search techniques that users take when completing the tasks.

We chose DBWipes because it is similar to commonly used visual exploration tools such as Tableau [2, 102] or Splunk [121] but specifically designed for solving the types of tasks in this study. Its integration with Scorpion means we do not need to train subjects in two separate systems and its web-based interface lets remote subjects participate without the need to install any software.

5.4.1 PARTICIPANTS

We recruited 13 participants that all have experience performing data analysis. 3 of the participants do not have a degree associated with computer science, 3 are graduate students in computer science, and the rest are data analysts or researchers at a European telecom company. Their experience with structured data analysis tools and their technical expertise vary from users that primarily use Excel to professional data research scientists. To evaluate their technical expertise, we asked subjects to self-rate their experience with SQL (as a proxiy for technical expertise) on a likert scale, with 1 being no knowledge of the language and 7 understanding how nested-queries, and group-bys work; the median score was 6, and the mean score was 4.8 because five participants self-rated a score of 4 or less (Figure 5-9). In terms of data analyst archetypes [62], the users with low expertise tended to be Scripters, with some working knowledge of programming languages such as Java or Python, and Application Users that primarily used GUI interfaces such as Excel. High expertise users tended to have or were pursuing advanced degrees in Computer Science. We labeled users with an expertise score < 5 as novices, and the rest as experts. Participants had never used the DBWipes nor the Scorpion interface, and few had direct experience with Tableau-like tools.

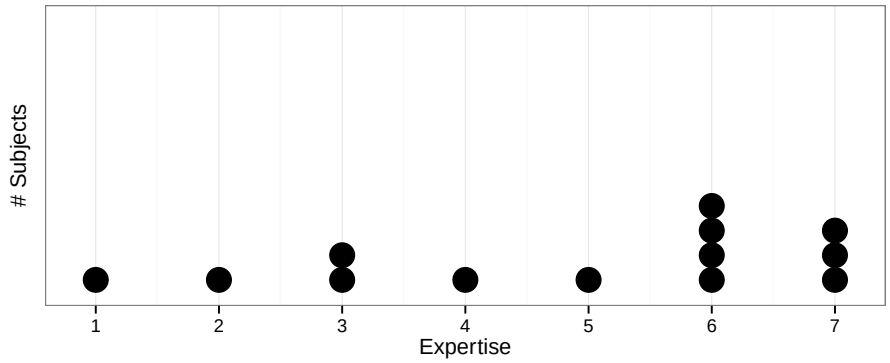


Figure 5-9: Distribution of Participant Expertise

5.4.2 EXPERIMENTAL PROCEDURES

We first asked participants to complete a pre-study questionnaire to state their demographic information and past experience with data analysis tools.

We then presented users with a three-part tutorial consisting of an introduction to the basic DBWipes tool (without scorpion), a verification task that tests the user's understanding of the interface, and an introduction to the Scorpion plugin. During this portion of the study, users could ask questions about the interface and we either referred the user back to the tutorial if it addressed the question or answered the question ourselves.

Following the tutorial, we asked users to complete three analysis tasks using DBWipes with or without Scorpion. Every user completed the same tasks, however the presence of the Scorpion tool was randomized. We also randomized the order that the tasks were presented to the user. In each task, we present the user with the visualized result of an aggregation query in the DBWipes interface, and specify a set of outlier aggregate values that we ask the user to explain.

Afterwards, the participants completed a post-study questionnaire and concluded with follow-up questions that the facilitator generated while watching the user during the study. When possible, we recorded the user's screen for the duration of the study.

5.4.3 TASK SPECIFICATIONS

The tasks vary in the type of aggregation query that we ask the user to explain, and the outliers in the underlying dataset. We designed two types of queries and two datasets for a total of 4 possible tasks. One of the possible tasks, described below, is ambiguous so we did not include it in the study.

Queries

The study uses two query templates that compute the total and average sales amounts for each day in the dataset.

SELECT day, sum(amt) FROM <table> GROUP BY day (Q5)

SELECT day, avg(amt) FROM <table> GROUP BY day (Q6)

The first query is designed to be easy to solve, because outlier values in the aggregation query ($sum(amt)$) are correlated with the cardinality of the attribute values, thus the anomalous attributes values are easily distinguishable in DBWipes' faceting interface. In contrast, the second query is designed to be challenging, because the aggregate values are not influenced by the cardinality of the attribute values and not discernable in the faceting interface. Our results show that this distinction affects the quality of the explanations that the users manually come up with.

Datasets

We generated three synthetic sales datasets for the study. One, called *simple*, is designed for use during the tutorial, and the others, called $hard_1$ and $hard_2$ are designed for the study tasks. The schema for the datasets are as follows:

```
sales(day int, state text, age text, gender text, amt float, id serial)
```

The domain of each attribute is as follows: *day* varies from 0 to 9, *state* is one of 41 US states, *age* is discretized into 4 categories, *gender* consists of M or F, *amt* is a positive floating point number, and *id* is a serially ordered primary key for the records. For *simple*, we reduced the cardinality of the state domain to 9 states:

$$
\begin{aligned}
day &\in [0, 9] \\
state &\in \{AL, AK, \ldots, WI, WY\} \\
age &\in \{< 18, 18 - 30, 30 - 50, > 60\} \\
gender &\in \{M, F\} \\
amt &\in \mathbb{R}^+ \\
id &\in \mathbb{N}
\end{aligned}
$$

The baseline data generation process creates $n \in \mathcal{N}(\mu_n, \sigma_n)$ records per state per day, where n is sampled from a normal distribution centered at $\mu_n = 50$ with a standard deviation of $\sigma_n = 5$. The value of the *amt* attribute $v_{amt} \in \mathcal{N}(\mu_{amt}, \sigma_{amt})$ is sampled from a normal distribution where $\mu_{amt} = 100$ and $\sigma_{amt} = 5$. The value of the other attributes are sampled uniformly at random from their respective domains.

We generated synthetic outliers for the task datasets so that the correct removal of the outliers would have a visibly noticable change in the visualized query results. To generate outliers, we apriori pick a set of attribute values and vary the parameters of the two above normal distributions. The "ground truth" values for each dataset is the aggregated value at day 0.

For $hard_1$, we increased the number of records for CA and MI during days 5 to 9 by a multiplicative factor $\mu_n = 100 \times (day - 4) \times 1.15$. In addition, we increased μ_{amt} for specific values of the *state* and *age* attributes using the following criteria:

$$\mu_{amt} = \begin{cases} \mu_{amt} + 50 & \text{if } state \in \{CA, MI\} \\ \mu_{amt} \times 3 & \text{if } state = FL \end{cases}$$

$$\mu_{amt} = \begin{cases} \mu_{amt} + 50 & \text{if } age = \text{`} > 60' \\ \mu_{amt} + 20 & \text{if } age = \text{`} < 80' \end{cases}$$

Using this criteria, there are numerous combinations of attributes that describe high *amt* values, however CA and MI are expected to dominate the total sales. We combined this dataset with Q5 as one of the study tasks. We did not consider Q6 because the dataset did not contain a clear set of outlier records that we could define as ground truth.

For $hard_2$, we generated outlier *amt* values during days 3 to 6 for the states MA and WA. For MA, we increased the mean value multiplicatively by $\mu_{amt} = \mu_{amt} \times 1.15 * (6 - |4 - day|)$ so that the value is maximized on day 4. For WA, we increased μ_{amt} by 20, and μ_n by 60. In this way, the number of sales in MA stay constant yet the amount of each sale increases significantly, which has an effect of influencing the *average* sales amount during the anomalous days. In contrast, the number of sales in WA increases greatly while the amount per sale increases modestly, so that it affects the *total* sales per day by a large amount. We combined this dataset with Q5 and Q6 to create two of the study tasks.

To summarize, each user was presented with a randomized ordering of the following three tasks: Q5×$hard_1$ (T1); Q5×$hard_2$ (T2); Q6×$hard_2$ (T3).

5.4.4 TASK INTERFACE

Figure 5-10 shows the interface for task T3. The top of the interface presents the task question on the left (A) and an answer form on the right. The grey points highlighted in the red rectangle (B) are the visualized outliers that the user is asked to explain. (C) shows a textual representation of the current candidate predicate that has been selected (D) in the faceting interface. The blue scatterplot (E) visualizes the results of executing Q6 on the records that match the candidate predicate. The user can add the candidate predicate as an answer by clicking on *"Add Filter to Answer"* (F).

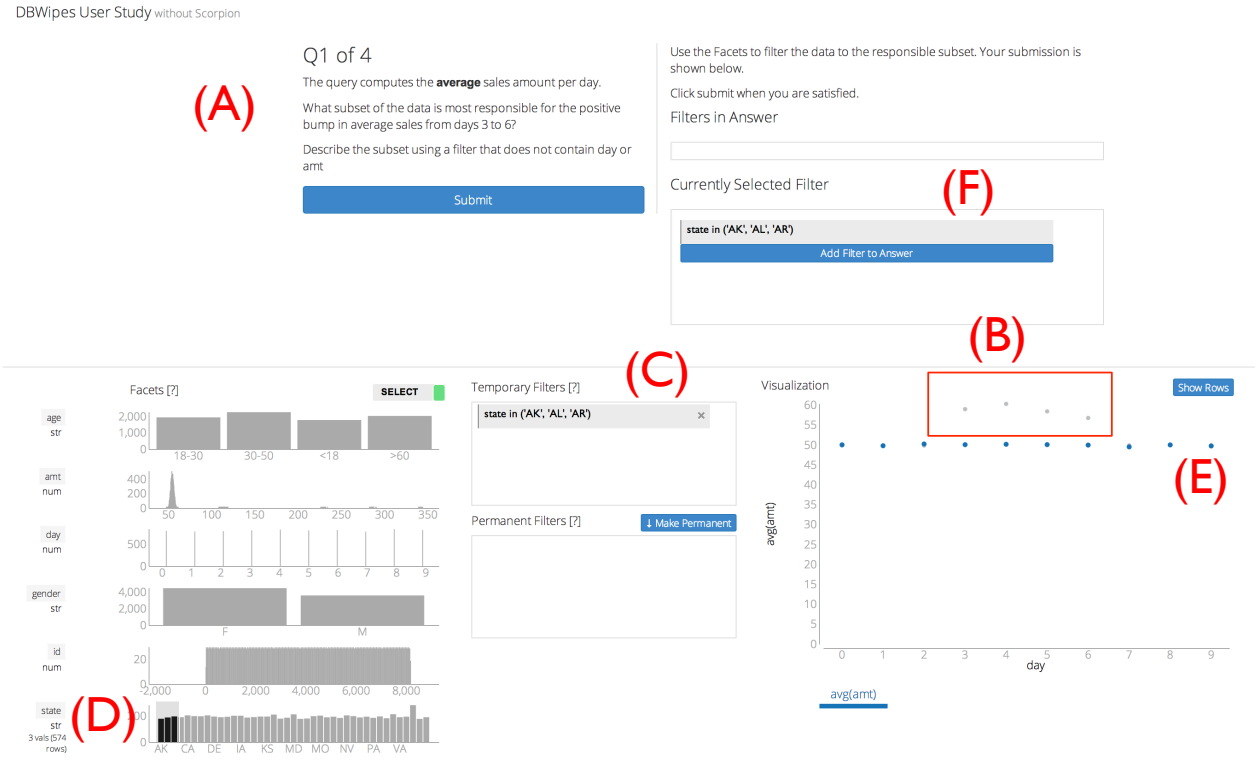


Figure 5-10: Task interface for task T3

5.5 QUANTITATIVE RESULTS

We used R and *lme4* [11] to perform a linear mixed effects analysis of the relationships between two dependent variables – task completion time and explanation quality – against the tool. As fixed effects, we used the tool (DBWipes with and without Scorpion), the task, and expertise (without interaction terms). As random effects, we used the intercepts for subjects. We ran the Levene test to check that the differences between the variances of the

dependent variables for each test condition (the heteroscedasticity) and found that they were not significant (> 0.66).

The task completion times were defined as the duration from the start of the task to when the user clicked submit, and were log-transformed to better approximate a normal distribution.

We computed the response scores from the amount that each aggregate value moved towards the true aggregate value given the user's explanation. Let p be the user's explanation (predicate), D be the task dataset, and v_i and v_i' be the aggregated value at day i over D and $\neg p(D)$ respectively. Furthermore, let d and g be the set of days with anomalous and normal results, respectively. Recall that v_0 is designed to be the true aggregated value of each day. Thus, we define the response score s_i for day i as:

$$s_i = \frac{|v_i - v_0| - |v_i' - v_0|}{|v_i - v_0|}$$

We defined a general response score that computes the average score for the outlier days and penalizes the amount that the results on the normal days deviate from their original values:

$$score_\alpha = \alpha \times \sum_{i \in g} \frac{s_i}{|d|} - (1 - \alpha) \times \sum_{i \in g} \frac{s_i}{|g|}$$

Since removing records from the dataset will inevitably have an effect on the result values, we use α to control the amount of penalization. When $\alpha = 1$, we only care about fixing the outlier days, whereas $\alpha = 0.5$ equally weights outlier and normal days. We report significance results for varying values of α

5.6 SCORPION REDUCES ANALYSIS TIMES

We found a significant main effect for tool ($p < 0.01$), a moderately significant effect for task ($p = 0.051$) and no significant effect for expertise ($p = 0.69$).

Task T1 was designed to have a distinctive "bump" in the facets that directly explains the outliers days $5 - 9$. Users easily found and tested the bump and were satisfied by the amount it affects the outliers. For this reason, the median task completion times were nearly equivalent between the two tools.

For tasks T2 and T3, the predicates were less obvious from the facets and Scorpion helped users complete the task $2\times$ and $1.3\times$ faster than those that answered the task manually. Moreover, the tasks were designed so the main outlier effects could be explained using

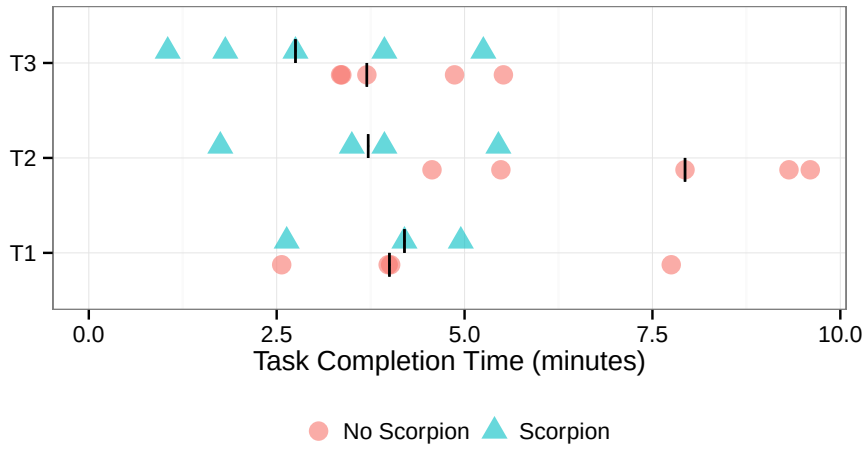


Figure 5-11: Task completion times for each task and tool combination.

single-attribute predicates. As the dimensionality of the explanation increases, we expect the Scorpion to have a much larger effect on completion times.

5.7 SCORPION IMPROVES ANSWER QUALITY

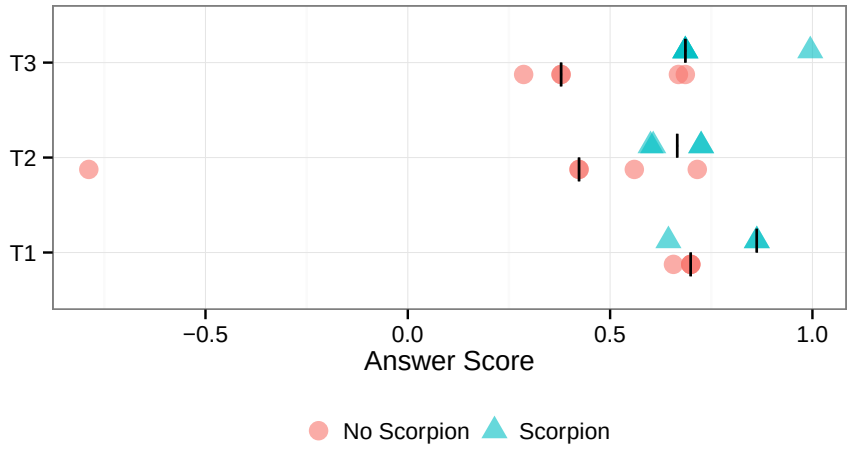


Figure 5-12: $score_1$ values for each task and tool combination.

Our analysis of $score_1$ values found a significant main effect for tool (0.021) a slight effect due to task (0.06) and no significant due to expertise (0.109). Figure shows the individual and

median $score_1$ values by task and tool. We found that Scorpion consistently finds predicates that explain the aggregated outliers.

For T2, the outliers are explained by the states $state \in \{MA, WA\}$, however only WA appears as an outlier in the faceting interface. Thus all but one user failed to manually identify the MA value. In contrast, every user that used the Scorpion interface submitted an explanation that contained both states. Similarly for T3, the outlier results are primarily explained by $state = MA$, which is not distinctive in the faceting interface. Thus, all but one of the manual solutions were misled by the high cardinality of $state = WA$ and chose it as the answer.

The very low $score_1$ for T2 without Scorpion was because the user selected $gender = M$ as the explanation, which reduced the outlier results by over twice their distance from the ground truth values, leading to a negative score.

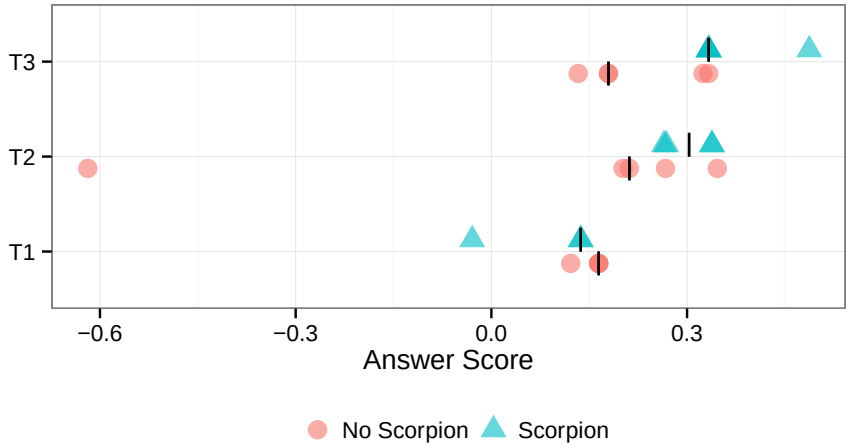


Figure 5-13: $score_{0.5}$ values for each task and tool combination.

As we reduce α, the significance of the tool's effect on $score_\alpha$ decreases. For example, when $\alpha = 0.8$, the tool has an effect with $p = 0.04$, and when $\alpha = 0.5$ (Figure 5-13), the tool does not have a significant effect $p = 0.11$. This is because Scorpion returns a list of explanations that vary in their effect on the outlier and normal results. Several users tended to pick the first explanation in the list, which has a large effect on every result value and thus penalized by the negative term in the scoring function.

We note that Scorpion does include more precise explanations that only effect the outlier values. A possible reason why users do not pick more precise explanations (when we measure the score using lower *alpha* values) may be because they do not cause a easily perceivable change in the main visualization and are regarded as uninteresting. A solution may be to

dynamically rescale the y-axis so that the changes to the outliers are significant. Alternatively, the interface could provide numerical scores that summarize the amounts that each predicate affects the outlier and non-outlier results.

5.8 SELF-RATED QUALITATIVE RESULTS

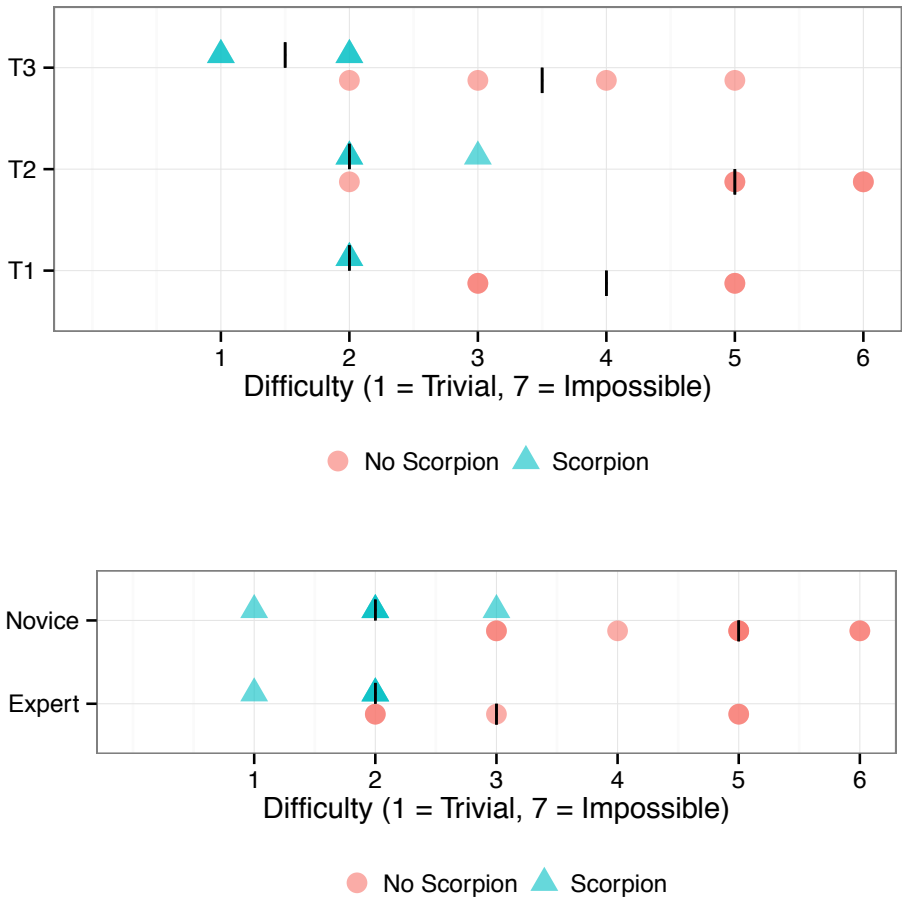


Figure 5-14: Self-reported task difficulty by task, expertise.

In the post-study feedback, we asked users to rate the perceived difficulty of each task. Figure 5-14 plots the reported task difficulties by task and expertise. We found a significant effect due to tool ($p < 0.001$) and expertise ($p = 0.008$) and no effect due to task ($p = 0.15$). When asked about the difficulty rating, a participant commented that it's *"probably impossible for human being to find the best answer... won't know if it's good or not"*. Others stated that

they *"wouldn't exhaustively try all combinations"*. We found that novice users perceived the greatest difference in difficulty between the two tools.

We then asked users to comment on the quality of explanations that Scorpion generated on a likert scale, where 1 is not useful and 7 is very useful. All users reported a rating of 5 or above, and the majority reported 7. One user noted, *"Instead of me doing the search, (Scorpion) presented a list of . . . best guesses."*

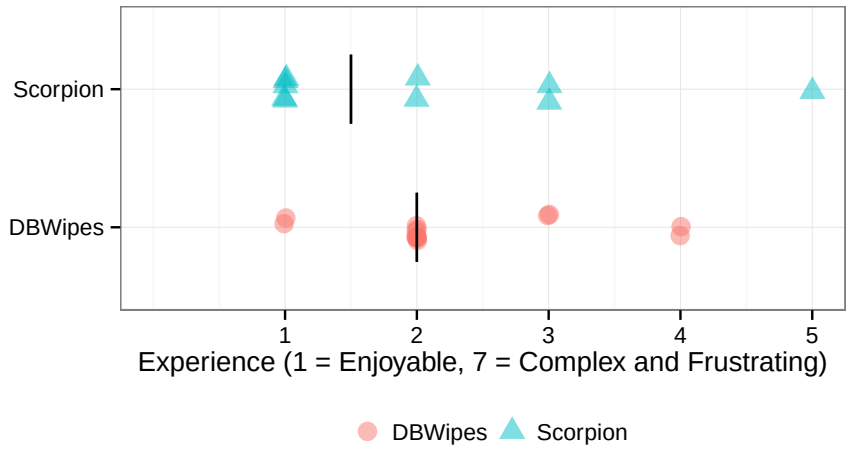


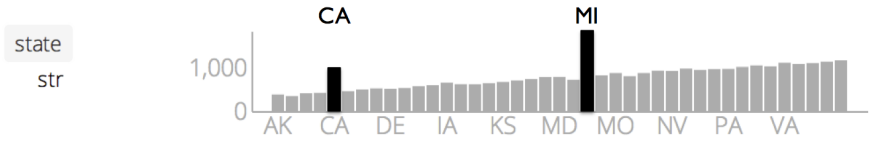
Figure 5-15: Self-reported experience using the tools.

Finally, we asked users to self-rate their experience using the baseline DBWipes tool and the tool with scorpion (Figure ??) on a likert scale where 1 is *enjoyable and easy to use* and 7 is *the interface was complex and frustrating to use*. Both tools were rated with a median of 2. One user rated Scorpion with a 5 because the λ slider component of the interface was difficult to understand, however she also noted that it was *"easy to solve all Scorpion tasks, because the tool is easy to use."*

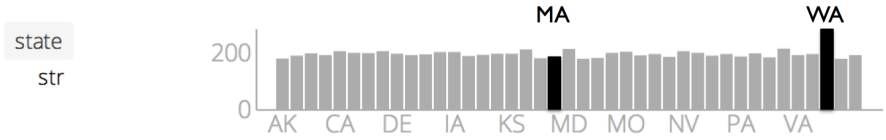
Despite these positive findings, we caution that these results should be taken with a grain of salt, because the selection process for gathering the participant population may bias the participants to those that are amicable to new tools such as Scorpion.

5.9 STRATEGIES FOR MINING EXPLANATIONS

We asked users to describe their strategy for solving each task. This section describes how users pick which combinations of attribute values to evaluate using each of the tools, how users evaluate a given explanation, and their confidence in their answers, and their impressions with the scorpion interface.



(a) Interface for T1.



(b) Interface for T2 and T3.

Figure 5-16: State facet interfaces (synthetic outliers highlighted in black.)

When users were asked to manually solve the tasks, the majority of the users systematically tried every attribute value one-by-one. Users first started with *age* because it was listed at the top, trying each age range individually, followed by *gender*, then finally *state*.

Almost all users exhaustively tested each individual *age* and *gender* value, and many users even tried all combinations of the two attributes, however few users exhaustively tested all 41 state values. Instead, users used the facets to look for skewed distributions (Figure 5-16) and focused on exploring the skewed regions (e.g., CA and MI for T1). Unfortunately, this led users down the wrong track for tasks T2 and T3, because only the state WA appeared as an outlier whereas MA was the dominant factor. As one user later noted, *"(I) thought I had a good shortcut by . . . looking for states that jumped out (in the facets) . . . turned out not a good idea because i missed a lot."*

Unfortunately, the number of possible combinations of attribute values is exponential in the cardinality of the attributes (5 ages $\times$ 3 genders $\times$ 42 states $=$ 630 total combinations). As one user commented, her strategy was to *"just try one by one, didn't try combinations, because the number of combinations would be a large number"*. Users quickly became fatigued when trying each state individually, and often gave up before finding the optimal predicate. One user said, *"I suppose as a human, I got bored."*

Most users used the amount that the normal results were affected as a proxy for the selectivity of the candidate predicate, and disregarded those that appeared to have low selectivity. Several users first filtered the visualization to only show the outlier days (i.e., used a permanent filter on the *day* attribute) and solved the task by examining how candidate predicates affected those days. This led to problems where the user spent a long time to pick a predicate that ultimately affected the normal days, leading to a low $score_{0.5}$.

When describing how they would approach similar tasks in practice, the users stated that they would use a similar strategy as that they used in the study. One user with experience with Tableau mentioned he would use it to solve the task, however when probed further, he say he would *"manually create filter widgets and . . . uncheck them one by one and see how they change the visualization. . . might try to create a DBWipes facet visualization."* A few expert users stated that they would write a program to try all combinations automatically and use a visualization similar to DBWipes to visualize the results.

Strategies Using Scorpion

Users used Scorpion to quickly generate a set of explanations (often within a minute). For most users, their subsequent strategy centered around Scorpion's top results. Some users immediately submitted the top suggestion, or tried the top several suggestions and submitted the one they most preferred – in both cases, the users cited that they trusted Scorpion's suggestions. This type of automation reliance [38] can potentially be unhelpful because Scorpion does not use any domain-specific information and may sometimes suggest attributes that represent dangerous or non-sensical real-world properties. Increasing the algorithmic transparency, such as explaining the attributes that have been explored or the evaluation criteria, can help assuage such over-reliance [70].

Other users spent the rest of the task evaluating and refining Scorpion's suggested results. As one user described, *"Scorpion's returned filters are at least a good baseline to understand what's going on. It saves the initial time that I would have spent clicking on a bunch of different filters."*

Only one user combined independent manual searching with Scorpion's suggestions. The user used Scorpion to identify a single dominant attribute, then explored subsets of the attribute to verify that the suggested filter was indeed influential. He then re-ran Scorpion with the attribute removed to find alternative recommendations and repeated the process until the suggested predicates did not adequately influence the outliers.

Most users were confused by the λ slider interface and either ignored it completely or set it to display the suggestions with the highest absolute impact. During the feedback, users mentioned that they would find it useful in real applications, however it was not needed in the study tasks.

Predicate Evaluation

We observed that users evaluated candidate predicates the same way irrespective of the aggregation function – they used the (non-negated) predicate to filter the dataset and

visually inspected the query results over the filtered data. *"if (I) saw it was similar, (I would) conclude that it was related to the outliers"*. Although this heuristic is accurate when the aggregation function is *sum*, it led to suboptimal results for Q6.

For example, the sales amount in the state WA are much higher than the average amount thus the visualization distinctly *"replicates the bump in the overall curve."* This misled many users to believe it has a similarly significant effect on the outlier values. In reality, its effect is minimal and the state MA most effects the outliers. Unfortunately, many users used this strategy for T3, which is why the difference in $score_1$ is most pronounced. The small number of users that used the *Select/Remove* slider to visualize the query results of the negated predicate were not misled.

Users care about the trade-off between the number of records that match a predicate and its effect on the outliers. For example, Scorpion includes a predicate for task T1 that matches one third of the states, which significantly reduces the total sales for all days. Users typically did not pick this predicate because *"(excluding) roughly a third of the states seems like it wouldn't be useful... (whereas) the alternative filter which only looked at MA also adequately explained the trend"*. Another user commented that *"some suggested filters where too broad, some too specific."* Thus, users ended up making a trade-off when picking which Scorpion result to pick.

Several users complained that the need to hover over a Scorpion result to view it in the visualization made it difficult to compare results. One user suggested rendering all of the Scorpion suggestions in the visualization so they can be directly compared. Several other users wanted some way to quantify the impact that each result has on the outliers, ideally *"in the same units as the aggregate measure"*. Most were satisfied with our suggestion to label each result with a value similar to $score_1$ used in this evaluation, along with its cardinality.

User Confidence

We asked users to describe their confidence in their answers and what would improve their confidence. For the manual tool, users consistently stated that systematically searching through all attributes combinations would increase their confidence, but that approach would take too long.

There were a number of reasons why Scorpion users expressed low confidence: one non-expert user forgot how the interface worked and was not confident that s/he was using it correctly; several others wanted to understand how the algorithm worked. In both cases, when we explained that the user used the interface correctly and how the algorithm worked, the users increased their confidence rating. This suggests that a Wizard interface [106] may

be appropriate for non-expert users, and additional information about how Scorpion searches for results would increase user confidence.

Other Scorpion users were confident in their answers. One non-expert user stated that *"in a big way, (Scorpion) was a confidence builder. Having some kind of algorithm generate (results) for you helps the confidence"*

5.10 CONCLUSION

This chapter introduced DBWipes, an interface for exploring and explaining anomalies in visualization that is integrated with the Scorpion outlier explanation tool described in Chapter 4.

Our user study finds that access to an automated explanation tool helps both novice and technical experts identify predicates that are correlated with anomalous visualization results in less time, and more accurately, than when performing the analysis manually. We also found that different aggregation queries require different search procedures, however users tend to employ a single manual heuristic that can lead to inaccurate or suboptimal results. The presence of an automated tool helps avoid these misconceptions and increases the confidence that users have in their explanations.

6 A Data Visualization Management System

The previous chapters described several self-contained components that each focus on a specific exploration task such as lineage querying or visual outlier selection, or outlier explanation. However, integrating these components into an existing data visualization system is challenging due to legacy architectural designs. For instance, the visualization rendering process is typically implemented as an imperative application (as opposed to a workflow) that is separate from data management, which makes integrating it with a lineage-tracking system difficult.

These challenges lead us to a natural question: "if we started from a clean-slate, how would a system that provides data management and visualization be designed?" This chapter presents the design of a Data Visualization Management System, which unifies the execution framework of a traditional database management system and a visualization system. Users specify data transformations and visualizations in a declarative visualization language that is compiled into a query execution plan that is primarily composed of relational operators and a small number of user defined functions.

Formulating the end-to-end visualization process as a relational query plan rather than an arbitrary imperitive program simplifies the task of tracking how input records flow through the plan and contribute to individual elements (e.g., a point in a scatterplot) in the visualization because we can leverage existing relational lineage tracking techniques. In addition, a unified visualization and data processing architecture has the potential to be both *expressive* via the high level visualization language, and *performant* by leveraging traditional and visualization-specific optimizations to scale interactive visualizations to large datasets.

6.1 INTRODUCTION

Most visualizations, including those described in this dissertation, are produced by retrieving raw data from a database and using a specialized visualization tool to process and render it.

At first glance, this decoupled approach makes sense because query execution appears to be a problem orthogonal to rendering and visualization. By connecting the two tiers with a SQL-based communication channel, the visualization community can focus on developing more effective visualization and interaction techniques, while advances from the data management community can transparently improve the performance of DBMS-backed visualization systems. In addition, certain operations, such as filtering the raw data for the subset within a visible bounding box, can be offloaded from the visualization client to the database.

However, it is increasingly difficult for this architecure to keep up with the growth of dataset sizes and the demand for more powerful exploration, annotation, and analysis features [46]. For example, in order to minimize the latency of user interactions, visualization tools will avoid roundtrips to the database by managing their own results cache and executing data transformations directly. We have identified the following key drawbacks of this architecture:

Provenance and Lineage Tracking

Foremost is the difficulty of tracking record-level provenance information across two different systems – the database management system and the visualization client, which is a necessary mechanism for many visual data analysis features, such as the explanation functionality described in Chapter 4. Although prior work have investigated efficient provenance tracking in database systems [44, 56, 113] and general workflow systems [8, 21, 41, 86] that decompose computations into a sequence of logical operators than can be reasoned about, visualization clients are typically implemented as a single imperative program whose provenance information is difficult to reason about.

Missed Optimization Opportunities

The database is unaware of visualization-level semantics and thus unable to perform higher level optimizations. For example, consider a dynamic slider that updates the paramater filtering predicate (e.g., "select * from sales where day = [slider value]") of a visualization. As the handle moves, the visualization will issue a large number of queries that only differ in the parameter value. However, the database is not aware of this fact, and will fully recompute each query and thus incur a significant amount of redundant computation.

Redundant Implementation

Visualization tools will often duplicate basic database operations, such as filtering and aggregation as a way to avoid the communication cost associated with sending those

operations to the database and retrieving the results. In addition, visualization developers will often re-implement common query optimizations such as r-tree [45] indexes and hash joins in order to ensure that the visualization responds quickly to user interactions. In fact, some tools even implement a custom database for this purpose [109].

Memory Constraints

Many visualization tools [19, 72, 111] assume that all raw data and metadata fit entirely in memory . These assumptions make these tools difficult to scale to larger datasets that exceed memory capacity.

6.1.1 A CLEAN-SLATE APPROACH

We propose to blend these two systems into a *Data Visualization Management System* (DVMS) that makes available all database features for the purposes of visualization. Our DVMS prototype, Ermac, embodies our two central ideas: a *declarative visualization language* that describes the mapping between raw data to the geometric objects rendered in the visualization, and a compiler that transforms a query in the language into a set of relational queries that are executed by a single query processing engine.

The relational formulation makes it feasible for provenance systems such as the one described in Chapter 3 to track individual tuples from an input source to the pixels rendered on the screen. Provenance support further enables advanced visual-analytic functionalities such as the ability to explain visualized outliers (Chapter 4).

This chapter describes Ermac's architecture, our current visualization language and compilation process, and the ECMAScript-based prototype implementation. The discussion in section 6.7 describes future research directions that are made possible by a unified visualization architecture.

6.2 OVERVIEW AND RUNNING EXAMPLE

Ermac is designed as a *data visualization engine*, meaning that it can be used as a standalone visualization system for data exploration, as the execution engine for a domain specific language within a general programming language such as ECMAScript or Python, or as the backend that executes specifications generated from visual direct manipulation tools such as Lyra [100]. Ermac takes as input a declarative visualization query, and performs the querying, data transformation, layout, and rendering operations to generate an interactive visualization.

Our key insight is that a significant portion of operations performed by a visualization system parallel those in the database system. For example, projecting data onto a coordinate system, calculating aggregate statistics, and partitioning the dataset into multiple views are all expressible as relational queries. Thus it should be possible to represent the end-to-end process of data transformation, layout, and rendering in relational terms as a single execution plan. This approach would confer the system with all of the benefits of a DBMS – heavily optimized operator implementations, data management, a cost-based optimizer, and secondary data-structures such as materialized views and indices.

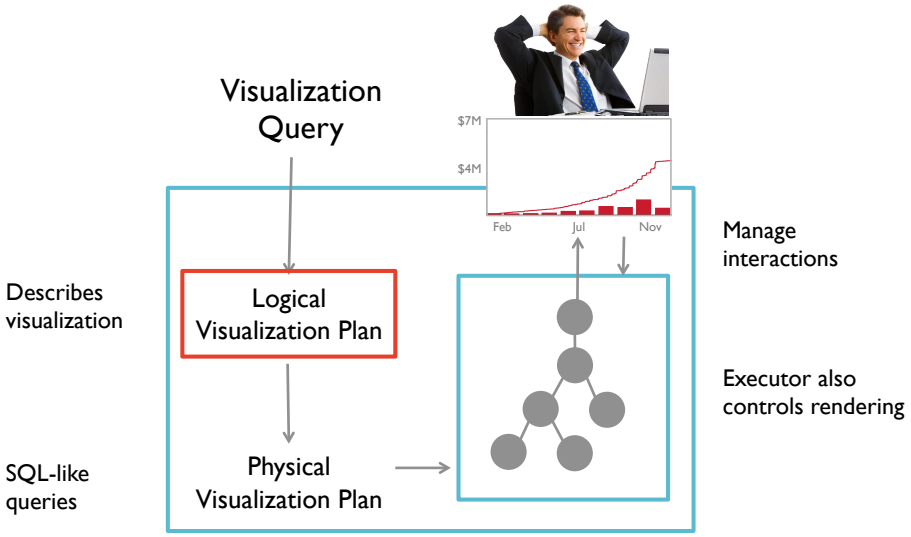


Figure 6-1: High-level architecture of a Data Visualization Management System

Figure 6-1 depicts Ermac's the high-level architecture. Ermac takes as input the user's visualization query and first compiles the query into a Logical Visualization Plan, or LVP for short (Section 6.3). The operators in the LVP describe high level steps such as mapping statistics to geometric objects, binning the data for a histogram, or computing quartiles for a boxplot visualization. Section 6.5 outlines how LVP is then optimized and further compiled into a Physical Visualization Plan (PVP) composed of *logical* relational operators such as join, filter, and project. The PVP finally goes through a traditional Selinger-style query optimization [101] step to produce the final *physical relational* operator plan that is executed to produce an interactive visualization. Ermac further manages the interaction between the visualization and the execution system. The businessman depicted in the upper right represents our model of a user that is satisfied after using the DVMS.

The following sections use the visualization in Figure 6-2 as the running example. The figure compares the weekly (bars) and cumulative (line) amounts that the Obama and Romney presidential campaigns spent in the 2012 US presidential election. The dataset is

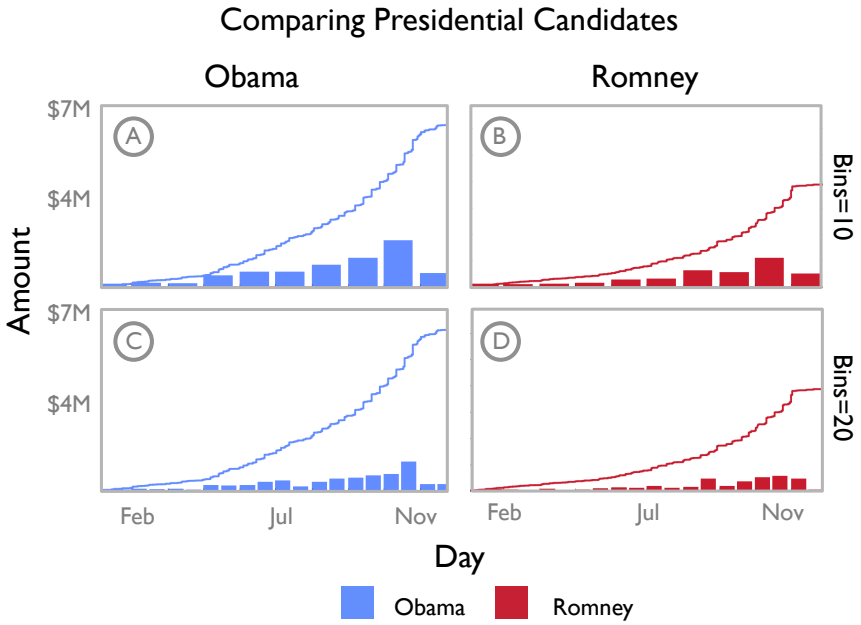


Figure 6-2: Faceted visualization of `expenses` table

provided by the Federal Election Commission [1]. The table attributes include the candidate name, party affiliation, purchase dates within a 10 month period (Feb. to Nov. 2012), amount spent, and recipient. We list the table definition below:

`election`(*candidate*, *party*, *day*, *amount*, *recipient*)

6.3 LOGICAL VISUALIZATION PLAN

A visualization is the result of a mapping from abstract data values into the visual domain. Ermac takes as input a visual specification that describes this mapping, and executes it on a relational table in the data domain to generate a set of visual elements rendered as pixels on the screen in the visual domain.

Figure 6-3 summarizes the process creating a simple visualization that compares Obama and Romney's expenses distributions. Each grey arrow represents a distinct processing step, and the arrows are primarily distinguished by whether they occur in the data domain (arrows 1 and 2), visual domain (arrows 4 and 5), or between the two (arrow 3 maps data onto visual objects).

[1] http://www.fec.gov/disclosurep/pnational.do

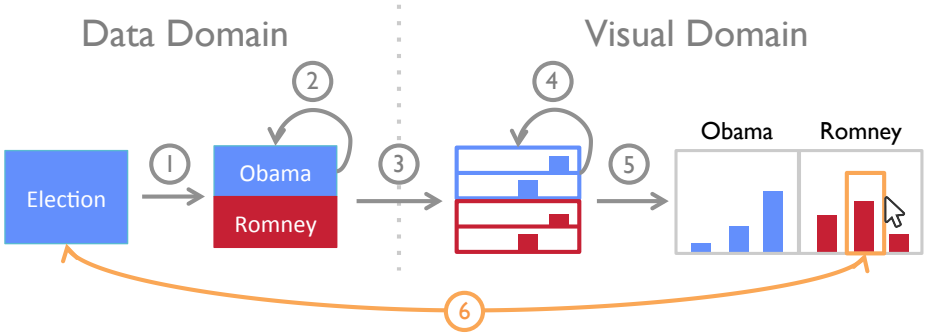


Figure 6-3: `expenses` Logical Visualization Plan.

For example, arrow 1 partitions the `election` table by candidate in order to compare statistics between the two, and arrow 2 computes data statistics such as the total expenses per week and the cumulative expenses by day. Arrow 3 maps data into visual attributes such as the x and y pixel coordinates and color (blue for Obama, red for Romney) – transformations in the subsequent operators are performed in visual domain. Arrow 4 performs positioning, layout, and visual transformations, and arrow 5 renders the final geometric objects (marks) in the visualization. The final orange arrow (6) represents visualization interactions, which trigger a complete or partial execution of a new visualization query.

Ermac currently borrows heavily from prior visual languages [111, 114] whose grammars decompose the above process into several orthogonal components[2]. For example, the components in layered grammar used by ggplot2 [111] include data to visual aesthetic mappings, statistical transformations, geometric objects, scales, and statistical transformations. The logical operator classes in the Logical Visualization Plan map almost directly onto the components in these grammars. The rest of this section describes each of our logical operators.

6.3.1 SYNTAX OVERVIEW

Our syntax is a nested list of clauses, where each `[class: operator]` clause describes the specific operator(s) for a given operator class. For example, `[geom: circle]` specifies that the geometric mapping should map attributes of the data onto properties of a circle mark, such as position, radius and color. Top level clauses define global operator bindings, and nested clauses are unique to a given `layer` (described below). Clauses may only be nested within `layer`, which cannot be nested within itself:

```
[class: operator]*      // top level clause
```

[2]We encourage interested readers to read those publications for an in-depth analysis of graphical grammars.

Class	Description
`data`	The input dataset(s).
`aesmap`	How attributes in the datasets are mapped to visual aesthetic attributes.
`stat`	Statistical transformations to apply to dataset.
`geom`	Which mark type to represent the visual aesthetics.
`pos`	Custom transformations to apply to the mark objects.
`facet`	How the data should be faceted along the x and y dimensions.
`scale`	Custom mapping from the data to visual domain.
`layer`	Add a new layer to the visualization

Table 6-4: Summary of classes.

```
[layer:                 // layer clause
   [class: operator]*   // nested clause
]*
```

An `operator` is defined by its name and an optional sequence of key-value parameter values. For example, `circle` defines the circle operator that uses default values for all of its properties, whereas `circle(radius:10)` defines the circle operator and sets the radius to 10 units. `class` clauses that perform data transformations, such as `stat`, also accept a sequence of `operator`s as input. As a shorthand, the operator name can be dropped for classes that only support a single `operator`, such as the `aesmap` and `facet` classes described in the next subsection.

```
operator = name | o | '[' o, operator ']'
o        = name([parameter: value]*) |
           [parameter: value]*
```

6.3.2 OPERATOR `class`ES

Ermac supports eight classes of operators, summarized in Table 6-4. The following subsection describes the function of each operator class and lists examples of its usage.

`data`

The `data` class specifies the input dataset for the visualization. Users can specify a relation in a database, a database query, a csv text file, or an array of attribute-value hashtables in the embedded programming environment. The following code snippet show examples of connecting to a database query and a web-based csv file.

```
data: db(url: 'postgres://...', query: 'SELECT ... ')
data: 'http://.../data.csv'
```

aesmap

The `aesmap` class specifies how attributes in the input dataset (e.g., amount, week) are mapped to visual aesthetics such as the x/y pixel coordinates and color. The user specifies a list of data attribute, visual attribute pairs.

facet

The `facet` class enables Ermac to render small-multiples [39] views in a two-dimensional grid. For example, Figure 6-2 partitions the election dataset by candidate and renders each partition using the same visual mapping side-by-side along the x axis. This class corresponds to arrow 1 in Figure 6-3.

stat

The `stat` class specifies the sequence of statistical transformations to run on the input dataset. For example, our current implementation supports arbitrary group-by aggregation, local regression (loess) smoothing, sorting, cumulative distributions, and box plots. This class corresponds to arrow 2 in Figure 6-3.

scale

The `scale` class defines the bi-directional function that maps values in the data domain to values in the visual domain. By default, Ermac uses a linear mapping for each attribute. For example, let the attribute `amount` $\in [0 - 100,000]$ be mapped to a position between $[5 - 100]$ pixels along the y-coordinate axis. The default linear transformation would be $y = amount/100000 * 95 + 5$. Alternative mapping functions include log transformations, or geographic coordinate projections.

geom

The `geom` class specifies which mark type should be used to render the input data and the algorithms for define the default layout positioning. For example, our current implementation supports circles (for scatter plots), lines, paths, rectangles (for bar charts or 2D-bins), text, and box-plots. This class corresponds to arrow 4 in Figure 6-3.

pos

The positioning class `pos` is analogous to `stat`, however it specifies transformations applied to marks in the visual domain (arrow 4). Re-positioning operations can vary from simple shifting transformations to offset text labels, to stacking curves on top of each other to create a stacked area chart:

```
pos: shift(dx:10, dy:30)
pos: stack
```

layer

`layer` is a special class that adds a new rendering layer to the visualization. Each layer can define custom marks, statistical transformations, and other `class` clauses that override the global defaults. Layers are rendered in their declaration order, so that the last layer is rendered on the top.

6.3.3 RUNNING EXAMPLE

Algorithm 3 Specification to visualize 2012 `election` data.

```
1: data: db(table: 'election', url: ...)
2: aesmap: x:day,y:amount
3: layer:
4:    stat: [sort(on:x), cumulative]
5:    geom: line
6: layer:
7:    stat: bin(bins:10)
8:    geom: rect
9:    //stat: bin(bins:VAR1)
10: facet:
11:    facetx: candidate
12:    facety: [VAR1 ← (10,20)]
```

Lines 1-5 of Listing ?? are sufficient to render a line chart that shows total cumulative spending over time during the 2012 US presidential election. The `data` clause specifies the input table (which may also be a SQL SELECT query), the `aesmap` clause specifies the aesthetic mapping from the *day* and *amount* attributes to the `x` and `y` positional encodings. The `layer` clause specifies that the statistical transformation should first sort the data by `x` (*day*) and then compute the cumulative sum over `y` (amount) for each day, and that the should be rendered using the line mark.

Lines 6-8 render a new layer that contains a histogram of the total expenditures partitioned by day into ten buckets. The `bin` operator partitions the `x` attribute into ten equi-width bins (i.e., months) and sums the `y` values (Figure 6-2.A).

It makes sense to compare the purchasing habits of the two candidates side-by-side (Figures 6-2.A,B). The `facet` clause (Lines 10-11) specifies that the data is partitioned by *candidate* name; the visualization draws a separate *view*, or subfigure, for each partition; and the views are rendered as a single row along the x (`facetx`) dimension.

It is often useful to compare visualizations generated from different operators or operator parameters (e.g., compare different sampling and aggregation techniques). Ermac's novel *parameter-based faceting* uses special dummy operators and parameters that are replaced at compile time. For example, Line 12 further divides the visualization into a 2-by-2 grid (Figure 6-2.A-D), where each row varies the `VAR1` operator in the specification. Thus, replacing Line 7 with 9 changes the `bins` parameter into a dummy variable that will be replaced with a binning value of either 10 (monthly) or 20 (bi-weekly), as dictated by Line 12.

6.4 DATA AND EXECUTION MODEL

Ermac's data model is nearly identical to the relational model, however we support data types that are references to rendered visual elements (e.g., SVG element). This allows the data model to encapsulate the full transformation of input `data` records to records of visual elements that are ultimately rendered[3].

For example, to produce the example's histogram, Ermac first aggregates the expenses into 10 bins, maps each bin (month) to an abstract rectangle record, and finally transforms the rectangle records to physical rectangle objects drawn on the screen. When the user specifies a faceting clause or multiple layers, Ermac also augments the `data` relation with attributes (e.g., `facetx, facety, layerid`) to track the view and layer where each record should be rendered. For example, the schema of the initial `data` relation for the running example would be:

`election`(*candidate*, *party*, *day*, *amount*, *recipient*, *facetx*, *layerid*)

Where the value of *facetx* is the same as the value of the *candidate* attribute. There are two values for *layerid*, one for each of the two layers in the specification.

Ermac additionally manages a `scales` relation that tracks the mapping from the domains of data attributes (e.g., *day*, *amount*) to the ranges of their corresponding perceptual

[3]The physical rendering is modeled as UDFs that make OpenGL/WebGL/HTML DOM calls

encodings (e.g., x, y pixel coordinates). For instance, our example visualization linearly maps the *day* attribute's domain ($[Feb, Nov]$) to pixel coordinates ($[0, 100]$) along the x axis. These records are maintained for each aesthetic variable in every facet and layer.

Representing all visualization state as relational tables lets Ermac compile each logical operator into one or more relational algebra queries that take the `data` relation and `scales` relation as input and update one of the two relations. For example, Ermac reads the `data` relation to update the attribute domains in the `scales` relation, whereas data-space transformations (e.g., `bin`) read the `x` (*day*) attribute's domain from the `scales` relation to compute bin sizes.

6.5 PHYSICAL VISUALIZATION PLAN

In this section, we describe how each of the logical visualization operators are compiled into SQL queries that compose the Physical Visualization Plan.

facet

When the `facet` operator is compiled, the downstream operators may also need to be modified to deal with dummy variables. The `facetx: candidate` clause (Line 11) partitions the data by *candidate* name and creates a unique facet attribute value for each partition. This is represented as a simple projection query:

```
SELECT *, candidate as facetx from data
```

The parameter-based faceting (Line 12) is compiled into a cross product with a temporary table, `facets(facety)`, that contains a record for each parameter value (e.g., 10 and 20):

```
SELECT data.*, facety.facety FROM data OUTER JOIN facets
```

Furthermore, `facet` replicates the downstream LVP for each `facety` value 10 and 20. If the `facetx` clause were also a parameter list of size M, the downstream plan would be replicated $2M$ times – once for each pair of `facetx, facety` values.

aesmap

The `aesmap` operator can be directly compiled into a projection query. For example, the clause on Line 2 can be compiled into:

```
SELECT day as x, amount as y FROM data
```

`stat` and `pos`

Although statistical and positioning transformations can potentially be arbitrarily complex, the majority of transformations can be modeled as one or more aggregation queries over the dataset. For example, computing the histogram on Line 7 can be described by the following query:

```
SELECT x, sum(y) as y FROM data
```

On the other hand, Line 4 computes a cumulative sum, which is difficult to express using traditional relational operators. Ermac currently compiles these operators into a user defined table function:

```
SELECT * FROM cumulative(data)
```

`scales`

Scale transformations are directly mapped into a projection query. Let *scalex* and *scaley* be the scaling functions for the x and y aesthetic attributes. Then the scaling query is simply:

```
SELECT scalex(x) as x, scaley(y) as y FROM data
```

`geom`

Each `geom` operator defines a Most `geom` operators can be modeled as a projection query. For example, Line 8 can be represented as a query that "fills in" the `data` relation schema a line mark's necessary attributes:

```
SELECT x, y, 'black' as color, 'solid' as dashtype FROM data
```

`layer`

Each layer can be considered a separate visualization that shares the same rendering viewport, faceting, and layout as the other layers. Ermac models each layer as a separate logical visualization plan that all share a common dummy source operator. When there is more than one layer, the compiler first replaces the `data` relation with the result of an outer join between the `data` relation and a temporary `layers(layerid)` table containing one record per layer:

```
SELECT data.*, layers.layerid FROM data OUTER JOIN layers
```

This ensures that the execution plan operates over a single input table, and the operators in each layer filter the `data` relation for the records with its corresponding layer id.

In order to ensure that visualization components, such as the axis scales, are consistent across the layers, the compiler injects synchronization barriers for operations that span across the layers. For example, Ermac learns the domains of the `scale` mapping functions by computing bounds of each attribute across all of the layers and facets:

```
SELECT min(x) as xmin, max(x) as maxx,
       min(y) as miny, max(y) as maxy FROM data
```

Rendering Operators

In addition to the above logical operators, Ermac also supports two types of rendering-specific logical operators. The first is for computing the layout of the visualization (e.g., position and bounding boxes for axes, headers, and plot), which is non-trivial to express in pure SQL. We represent this operator as a user defined table function over the `scales` relation:

```
SELECT * FROM layout(scalestable)
```

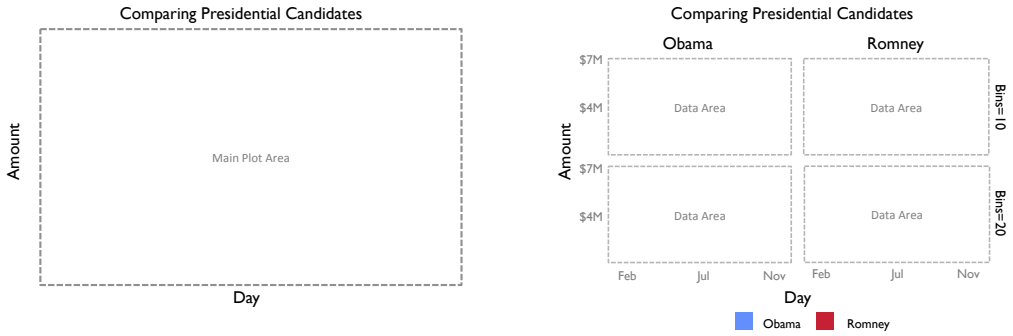
The second type are the operators for actually rendering mark objects to the viewport. Figure 6-5 depicts the rendered result after each of the three rendering operators. The first (Figure 6-5a) renders visualization level components that are independent of the `data` relation, such as the headers, axis titles and the main plotting area. The second (Figure **??**) renders non-mark components that are derived from the `data` relation, such as the plotting area for each facet and layer, as well as facet headers and axes. The final rendering operator iterates through the `data` relation and renders each mark object into its corresponding plotting area.

The first two rendering operators are implemented as user defined table functions, whereas the latter is simply a projection using a user defined rendering function:

```
SELECT render_svg(*) FROM data
```

6.5.1 DISCUSSION

Although we have developed compilation strategies for all major logical operators, many of the relational queries rely on expensive cross-products or nested sub-queries. Many of these operations are unavoidable, regardless of whether Ermac or another system is creating the visualization. However, by expressing these expensive operations declaratively, we can



(a) First Rendering Pass

(b) Second Rendering Pass

Figure 6-5: Visualization after each rendering operator

use existing optimization techniques and develop new visualization-informed techniques to improve performance.

For instance, Ermac knows that queries downstream from parameter-based faceting will not update the `data` relation so it can avoid redundant materialization when executing the cross-product. Identifying further optimizations for individual and across multiple LVP operators poses an interesting research challenge.

6.6 IMPLEMENTATION

Ermac is currently implemented as a CoffeeScript/ECMAScript workflow execution engine that takes as input a JSON-encoded visualization specification, renders the visualization as a Scalable Vector Graphics (SVG) object, and returns an ECMAScript object that contains the SVG, a table containing the references and attributes of the visualized DOM elements in the visualization, and a table containing the scales and layout metadata. Visualization queries are compiled into a directed-acyclic-graph of the logical operators described in Section 6.3. Rather than compiling the logical operators into SQL queries, Ermac directly transforms them into a physical relational operator graph. The physical operators can run in the browser as well as on a Node.js server, and the executor supports split execution where different subsets of the workflow can be executed on either location. Figure 6-6 shows a gallery of visualizations that Ermac can render from a randomly generate dataset. These examples illustrate different types of faceting, geometric objects, statistical aggregations, and layering that the system can support.

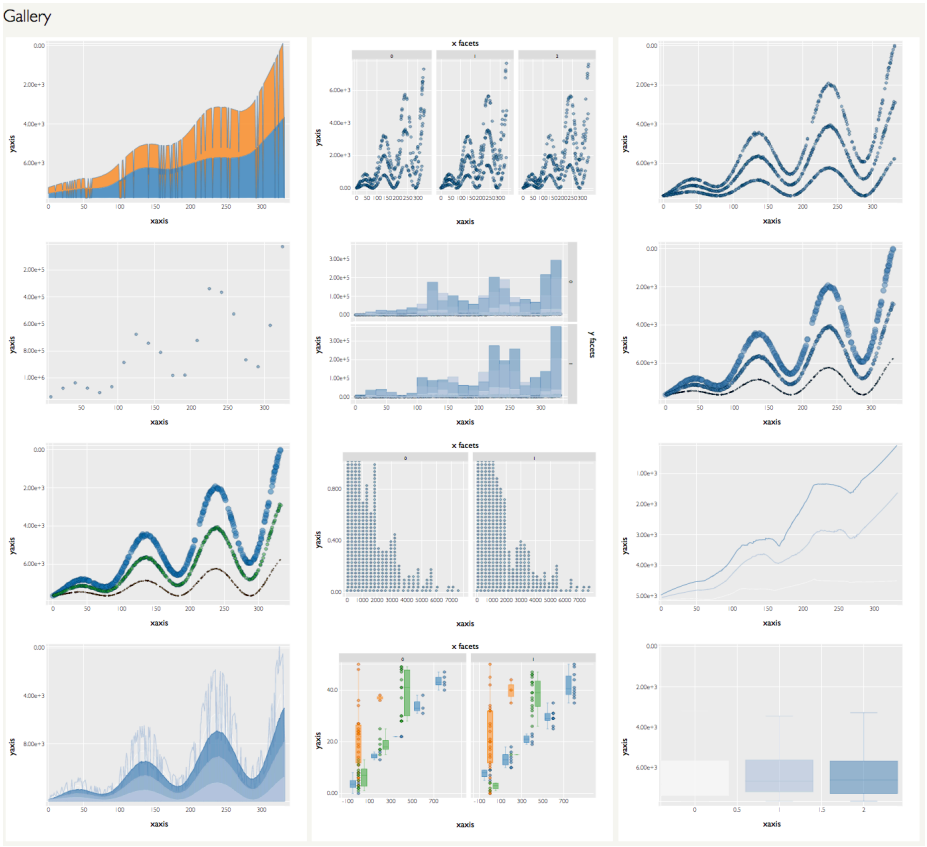


Figure 6-6: Gallery of Ermac generated visualizations.

6.6.1 OPERATOR IMPLEMENTATIONS

All Ermac operators are subclassed from a generic `Node` operator. The operator takes as input the pair of `data` relation and `scales` relation, and exposes the following simple interface:

```
class Node
  constructor: (@params={}) ->

  # @private
  # Private method that prepares inputs before calling compute()
  run: () ->

  # @public
  # Subclasses override this function
  compute: (datatable, scalestable, params, callback) ->
```

Both physical and logical operators override the `compute()` method, and the private

`run()` method validates, prepares and partitions the input data before calling `compute()` one or more times. Ermac provides generic implementations of each logical operator that shields the developer from dealing with validation and preparation. Custom logical operators simply specify a minimum input schema that the `data` relation must adhere to (e.g., x, and y attributes must be present in order to render a point) and an optional list of attributes as the partitioning key. `run()` partitions the pair of `data` relation and `scales` relation; each `compute()` call takes as input the pair of partitions with the same key value.

Partitioning is necessary for correctness – the faceting clause in the specification imposes a grid-like structure on the output visualization. Each operator may operate on partitions of the `data` relation that pretain to a given row, column or individual sub-plot in the grid, or on the full `data` relation. For instance, the x and y axes are typically rendered consistently across the sub-plots, so their domain information should be computed across all of the data. In contrast, statistical summaries such as cumulative distributions are computed for each sub-plot in isolation.

Within the `compute()` method, developers interact with Ermac tables using a method chaining syntax similar to the syntax in DryadLinq [120] and Spark [122]. These calls build an internal query plan that Ermac executes when the operator accesses data in the table, or at operator boundaries. For example, the following CoffeeScript code snippet filters the `data` relation where $x > 10$ and joins the result with the `scales` relation:

```
datatable
  .filter((tuple, idx) -> tuple.get('x') > 10)
  .join(scalestable, ['facetx', 'facety'])
```

We have implemented projection, filter, cross-product, outer-join, limit, offset, orderby, union, and partition operations. In addition, Ermac can internally represent tables in columnar and row formats, as well as partitioned on a set of table attributes. The latter representation is beneficial because nearly every operator first partitions the `data` relation by a combination of the $facetx$, $facety$, and $layer$ attributes.

6.6.2 USAGE

The following code snippet creates the visualization in Figure 6-2. With the exception of string quotes and formatting differences, the specification is nearly identical to the syntax presented in Section 6.3. The `ermac()` call returns a compiled visualization object and the `render()` statement simply renders the visualization within the specified DOM element.

```
plot = ermac(
  data: election
  aes:
    x: 'day'
    y: 'amount'
  layer:
    stat: [ { type: 'sort', on: 'x' }, 'cumulative' ]
    geom: 'line'
  layer:
    stat: { type: 'bin', bins: 'DUMMY' }
    geom: 'rect'
  facet:
    x: 'candidate'
    y: { type: 'DUMMY', vals: [10, 20] }
)
el = null;  // initialize to a DOM element
plot.render(el)
```

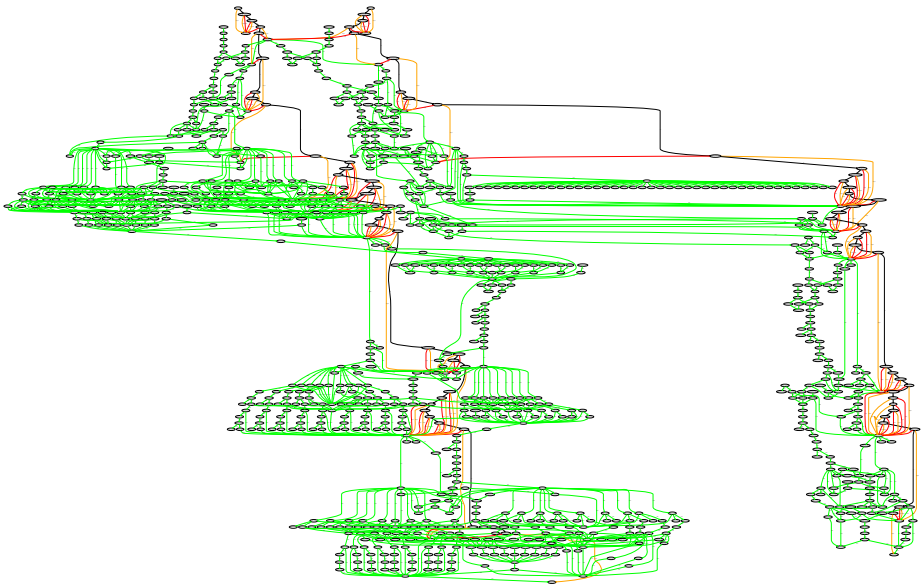


Figure 6-7: Workflow that generates a multi-view visualization

Finally, Figure 6-7 shows an example of the compiled workflow. The black arrows connect

the physical operators in the workflow, the green arrows connect table operations, and the red and orange arrows connect a physical operator with its input and output `data` relation, respectively. We note that the green edges represent a single set of relational transformations from the input data to the resulting visualization.

6.6.3 INTERACTION

Ermac visualizations support hovering over, selecting, and clicking on elements in the visualization. Developers can register callback functions for `select`,`hover`, and `click` events, and Ermac passes the active view and a set of tuples containing the corresponding visual elements to the callback function. For example, the following code fragment registers a `select` event handler that prints the id of each visual element that the user selects:

```
plot.on("select", (tuples, view) ->
  tuple.each (tuple) -> console.log(tuple.get('id'))
)
```

6.6.4 OPTIMIZER

Ermac currently preforms a very simple set of rule-based optimizations. The operator placement algorithm assumes that the client and server have identical performance and places operators to minimize the amount of network traffic, with the constraint that the rendering operators must be on the client. The cache placement algorithm currently inserts a caching operator immediately before the first rendering operator so that subsequent executions of the plan (e.g., in another user's browser window) can avoid executing a significant portion of the plan and simply directly render the cached data from a pre-computed file. The operator merging algorithm combines The `compute()` method of adjacent operators that share the same partition key in order to avoid unnecessarily repartitioning the `data` relation.

6.6.5 PROVENANCE

One of the key reasons that Ermac composes the large relational plan shown as the green edges in Figure 6-7 is to simplify the task of tracking provenance (and lineage) information. This allows Ermac to employ the techniques described in Chapter 3 to manage this provenance information.

Our current implementation uses a barebones provenance system that tracks operator provenance (the graph in Figure 6-7) and record-level provenance (the input records of each operator that contributed to each output record) and provides a simple provenance query interface to query the provenance of operators and records. The operator provenance graph is modeled as a directed graph where child operators consume the results of parent operators. Ermac provides standard graph traversal functions for accessing parents, children, ancestors, and descendents.

The record-level provenance interface supports backward queries of the form "what input records of operator A contributed to a subset of operator B's output records?" and forward queries of the form "what output records of operator A contributed to a subset of operator B's input records?":

```
ermac.backward(records or record ids, A, B)
ermac.forward(records or record ids, A B)
```

Record-level provenance queries return a collection of records that can be manipulated as a native ECMAScript array. For example, let *marks* be a set of marks in Figure 6-2's view A that are selected by the user. The following code snippet first retrieves the input records at the Source operator that contributed to *marks*, finds the marks in view B that share the same inputs, and iterates through the resulting marks to highlights each one.

```
inputs = ermac.backward(marks, ViewA, Source)
marks  = ermac.forward(inputs, Source, ViewB)
marks.each((mark) -> mark.highlight())
```

The code below extends the visualization example in Section 6.6.2 with a brushing and linking interaction between `plot` and a second visualization of the election data, `plot2`. When the user selects data in `plot`, the visualization elements in `plot2` are also highlighted. The selection handler executes a backward lineage query to retrieve the input tuples of the selected visual elements, and a forward lineage query to find all of the visual elements derived from those inputs. The final line highlights each of the visual elements.

```
// plot2 is a second visualization of the election data
plot2 = ermac(...);

plot.on('select', (tuples, view) ->
  inputs = ermac.backward(tuples, view, 'source')
  marks = ermac.forward(tuples, 'source', null)
  marks.each((mark) -> mark.highlight())
)
```

6.6.6 FINE TUNING

Creating a visualization goes beyond computing and rendering a graphical layout. It also involves typography (e.g., the typeface, the font size), the use of whitespace within and between subplots, the choice of color, and other fine-tuning elements. Although these details are not the focus of the system, Ermac implements two presentation-related features to make its visualizations more pleasant and configurable.

First, users can use cascading style sheets (CSS) to tune each of the graphical elements in the visualization. The default Ermac stylesheet mimics a style similar to ggplot2, with a light grey plot background, white grid lines, and subdued saturation.

Second, Ermac implements a simple constraint-based system to intelligently format, resize, and position axis and facet labels. The primary purpose is to improve legibility by avoiding labels that overlap with each other or exceed the side of its bounding box, and to adhere to aethestic design principles such as making appropriate use of whitespace. The system supports text transformations such as font size reduction, truncation, rotation, and can hide labels or resize the graphical plotting area as a final resort.

6.7 BENEFITS OF A DVMS

Dava visualization is part of a larger data analysis process. Although we have proposed techniques for a DVMS to manage the data transformation, layout, and rendering processes for creating static data visualizations, the vision is for an interactive DVMS system that manages how data is viewed, explored, compared and finally published into stories for consumers to experience.

To this end, there are numerous interesting research directions to explore, such as (1) expanding our language proposal (Section 6.3) into a comprehensive language that can

also describe user interactions in a manner that is amenable to cost-based optimization, (2) understanding interaction and visualization-specific techniques that can be used in an optimization framework to either meet interactive (100ms) latency constraints or mask high-latency queries, (3) exploiting different classes of hardware (e.g., GPUs) that are optimized for specific types of visualizations, and (4) incorporating recommendation and higher-level analysis tools that help users gain *sound* insights about their data.

The rest of this section outlines some immediate steps that help address each of these research directions.

6.7.1 VISUALIZATION FEATURES

Lineage-based Interaction

Many visualization tools provide provenance tracking as a graph of historical actions and/or states [40, 47, 60], or a simple undo log. In contrast, a DVMS can track how individual data records are transformed during the visualization rendering process, and how visual elements change and are manipulated as the user interacts with the visualization. This functionality can potentially increase the richness and performance of visualization interactions.

As one example, consider Brushing and linking [13] which is a core interaction technique (Figure 6-3 arrow 6) where user data selections in one view are reflected on the corresponding data (by highlighting or hiding them) in the other views. To do this, selected elements must be traced back to their input records, and then forward from those inputs to the visual elements in the other views. Unfortunately, existing visualization tools either require users to track these lineage relationships manually [19, 108], or provide implementations that often scale poorly to larger datasets and more complex visualizations.

In contrast, the DVMS' relational formulation captures these lineage relationships automatically, and can thus express brushing and linking as lineage queries. Furthermore, workflows allow the DVMS to optimize and scale interactions to very large datasets with little user effort. For example, the DVMS can automatically generate the appropriate data cubes and indices to optimize brushing and linking similar to the techniques used in imMens [72] and nanocubes [71].

The database community has explored many lineage optimizations [43, 56, 64, 118], however, additional techniques such as pre-computation and approximation will be necessary to efficiently support a truly interactive visualization environment.

Visualization Estimation and Steering

Users can easily build workflows that execute slowly or require significant storage space to pre-compute data structures, and it would be valuable to alert users of such costs. The DVMS can make use of database cost estimation [28, 29, 101] techniques to inform users of expensive visualizations (e.g., a billion point scatterplot) and inherent storage-latency trade-offs, and to steer users towards more cost-effective views. The latter idea (e.g., query steering [22]) may benefit from understanding the specification that produced the queries.

Rich Contextual Recommendations

Recommending relevant or surprising data is a key tool as users interactively explore their datasets. Prior work has focused on recommending visualizations and queries based on singular, but semantically different features such as data statistics [76], image features [87], or historical queries [65, 89, 98]. A DVMS can control and use all of these features to construct more salient recommendations to the user. For example, image features such as mountain ranges may be of interest when rendering maps, whereas the slope of a line chart is important when plotting monthly expense reports.

Result analysis

Several recent projects [80, 115], including the Scorpion project described in Chapter 4 extend databases to automatically *explain* anomalies and trends. Thus the DVMS can use these extensions "for free" to not only *present* data, but also embed functionality to automatically *explain and debug* the results. Chapter 5 explores how these algorithms can be integrated into an exploratory visualization system.

6.7.2 QUERY EXECUTION

Developing visualizations that are interactive across various environments and client devices (e.g., phone, laptop) can be challenging. The DVMS can allow users to specify latency goals (e.g., 200ms interaction guarantees) and use end-to-end optimizations to satisfy these constraints.

Rendering Placement

Rendering placement decides where to render visualizations given the client's available resources. For instance, heatmaps may be faster to render server-side and send to the client

as a compressed image, whereas histograms are faster to send as data records and render on the client.

Psychophysical Approximation

Psychophysical approximation computes approximations of the visualization in a way that minimizes user perceived error, and is widely used in image and video compression. For example, humans are sensitive to position but have trouble discerning small color variations. DVMSes can then respond to poor network bandwidth by pushing down an aggregation operator to coarsely quantize the color of a heatmap to match a smaller data type (e.g., `short` instead of `long`), and thus reduce the bandwidth demand by 4×. Alternatively, the system can aggregate the histogram data into coarse bins and use pre-computed data structures to reduce latency. Developing sufficient annotations to automate this optimization is an interesting research direction.

Visualization Materialization

The DVMS could use materialization techniques to pre-compute entire visualizations or components of the execution plan. This can be valuable when publishing visualizations to a consumer audience that expects low latency interactions but does not want to modify the visualization specification. This can be coupled with view maintenance techniques to, for example, update the visualization as the underlying dataset changes, as is the case in data streams. Alternatively, modifications made in the visualization, in applications such as data cleaning, can be transparently propogated as updates to the dataset.

6.8 CONCLUSIONS

The explosive growth of large-scale data analytics and the corresponding demand for visualization tools will continue to make database support for interactive visualizations increasingly important. We proposed Ermac, a Data Visualization Management System (DVMS) that executes declarative visualization specifications as a series of relational queries. The following two chapters focus on the implementation of a provenance-based outlier analysis feature that can be integrated into a DVMS such as Ermac. Section 6.7 describes exciting future research directions that exploit the DVMS' unified execution model to enhance the functionality and performance of a DVMS.

7 Related Work

The projects described in this dissertation each addressed a distinct problem in the design of a general visual exploration and analysis system, and the corresponding topics span the visualization, database, and data provenance communities. This chapter discusses the past work that this thesis builds upon, as well as more recent developments since the publication of the papers presented in this thesis.

The following sections are organized as follows: Section 7.1 provides an overview of data visualization systems, Section 7.2 describes prior work on provenance systems and the relevant theory, and Section 7.3 introduces techniques for analyzing data analysis results.

7.1 DATA VISUALIZATION SYSTEMS

Previous work in visualization systems have traded-off between expressiveness and performance. For instance, popular toolkits such as D3 [19], protovis [18] and matplotlib [51] are highly expressive, however they require low-level programming that impedes the ability to quickly iterate and do not scale to large datasets. Declarative grammar-based languages such as the Grammar of Graphics [114] and ggplot2 [111] are expressive domain-specific languages designed for rapid iteration, however they do not scale beyond their host environments of SPSS and R.

Recent systems address these scalability limitations by either adopting specific data management techniques such as columnar data representation [63], pre-computation [72], indexing [71], sampling [4], speculation [61], and aggregation [12, 112], or developing two-tiered architectures where the visualization client composes and sends queries to a data management backend [57, 102]. The former approaches are optimized towards properties of specific applications or visualization types and may not be broadly applicable. The latter forgo the numerous cross-layer optimizations described in Section 6.7.

7.2 PROVENANCE MANAGEMENT SYSTEMS

There is a long history of provenance and lineage research both in database systems and in more general workflow systems. There are several excellent surveys that characterize provenance in databases [30] and scientific workflows [17, 36]. In this section, we survey prior provenance systems work in terms of general workflow systems, database systems, and other systems.

7.2.1 WORKFLOW LINEAGE

Most workflow systems support custom operators containing user-designed code that is opaque to the runtime. This presents a difficulty when trying to manage cell-level (e.g., array cells or database tuples) lineage. Some systems [41, 69] model operators as black-boxes where all outputs depend on all inputs, and track the dependencies between input and output datasets. Efficient methods to expose, store and query cell-level lineage is still an area of on-going research.

Several projects exploit workflow systems that use high level programming constructs with well defined semantics. RAMP [53] extends MapReduce to automatically generate lineage capturing wrappers around Map and Reduce operators. Similarly, Amsterdamer et al [9] instrument the PIG [88] framework to track the lineage of PIG operators. However, user defined operators are treated as black-boxes, which limits their ability to track lineage.

Newt [73] is a provenance system for HyRacks [16] that also provides a lineage API for custom operators. Unlike SubZero, the operator code makes separate `addInput(record, tag)` and `addOutput(record, tag)` calls and dependency relationships are defined between all input and output records with the same tag value that also obey temporal causality i.e., output records can only depend on inputs that were registered beforehand. Their API may be easier to use because the system can infer lineage relationships on behalf of the developer. In addition, Newt always materializes lineage information and does not provide mechanisms nor policies to manage materialization strategies..

Other workflow systems (e.g., Taverna [86] and Kepler [8]), process nested collections of data, where data items may be images or DNA sequences. Operators process data items in a collection, and these systems automatically track which subsets of the collections were modified, added, or removed [10, 82]. Chapman et. al [27] attach to each data item a provenance tree of the transformations resulting in the data item, and propose efficient compression methods to reduce the tree size. However, these systems model operators as black-boxes and data items are typically files, not records.

7.2.2 DATABASE LINEAGE

Database systems execute queries that process structured tuples using well defined relational operators, and are a natural target for a lineage system. Cui et. al [33] identified efficient tracing procedures for a number of operator properties (Section 3.6 describes several mechanisms that can implement many of these procedures.) These procedures are then used to execute backward lineage queries. However, any language that allows custom operators will need to deal with user defined operators and their model does not allow arbitrary operators to generate lineage, and treats them as black-boxes.

Trio [113] was the first database implementation of cell-level lineage, and unified uncertainty and provenance under a single data and query model. Trio explicitly stores relationships between input and output tuples, and is analogous to the full provenance approach. Ikeda et. al [52, 55] extended the Trio work and explored the relationships between SQL and lineage. They defined the semantics of logical provenance, a special case of the mapping lineage described in Section 3.6.2, and use the semantics to statically construct backward mapping functions for a useful class of SQL Select-Project-Join (SPJ) queries.

Allison et. al [116] introduced the notion of a weak inverse function. Such a function can only approximately compute the lineage of a given subset of an operator output, and requires an additional *verification function* that has access to the input array values to accurately compute the lineage [1]. Although inefficient, SubZero's payload lineage can model weak inverse and verification functions by encoding the output and input array values inside the binary payload and implementing both functions in the payload function. It is interesting to consider an efficient intermediate lineage representation similar to weak inverse functions that lay between mapping lineage, which is too restrictive, and payload lineage, which is too general.

7.2.3 PROVENACE IN OTHER SYSTEMS

The SubZero runtime API is inspired by the PASS [84, 85] provenance API. PASS is a file system that automatically stores and indexes provenance information of files and processes, and provides a powerful provenance query interface. Applications can use the *libpass* library to create abstract provenance objects and relationships between them, analagous to producing cell-level lineage. PASS is primarily focused to tracking the relationships between process execution and file-level (coarse-grained) modifications to the file system. SubZero extends this API in the context of fine-grained lineage support in scientific applications.

[1] In their work, they assume the weak inverse function has access to output values. In contrast SubZero assumes the mapping function only has access to cell coordinates.

Provenance has also been extended in the declarative networking community. Declarative networking [75] models network protocols as recursive queries over distributed relational state. The network datalog (NDLog) language extends Datalog [74] to be aware of network-related constraints on distribution, communication, and state. ExSpan [124] models provenance information as distributed tables that track the dependencies between tuples (state) and NDLog rules, and develop incremental materialization rules for maintaining these provenance tables as an NDLog program executes. Subsequent work on the Y! [119] system use counterfactual logic to support "Why not?" provenance queries that ask why an expected tuples does not exist, for example, why there are a *lack* of requests in the network.

Finally, information flow control in operating systems [14, 68, 123] tracks how data flows within the application or OS in order to control data sharing with the external world. In addition, systems such as Retro [67] track systems level provenance (called an action history graph) and uses it to undo undesirable historical actions, then selectively re-run legitimate actions that depended on undoed actions. Warp [25] extends the Retro model to database-backed web applications by tracking how the web application updates and reads the database state.

7.3 OUTLIER EXPLANATION

The topic of deriving the relationships between the output of a computation function and the function inputs has been explored in numerous domains. This section focuses on prior work in the context of database queries and how the same problem maps to closely related domains such as network analysis.

7.3.1 SENSITIVITY ANALYSIS

Sensitivity analysis studies how uncertainty or variance, of inputs to a model relate to the uncertainty or variance of the output values. Saltelli [95] presents an overview of the area. The *why explanation* problem can be modeled as a sensitivity analysis problem, where the SQL query is the model, and we want to understand in what ways the aggregation results are sensitivy to differente subsets of the database. The main differences are that the input is the entire database state so more efficient methods are necessary to make this problem even remotely tractable, and that we are interested in a specify type of change in the output (e.g., average temperature should be lower) rather than a general analysis of the output variance.

7.3.2 OUTLIER DETECTION

Outlier detection is a core technology in applications as diverse as video processing to detect intruders, industrial manufacturing to identify defective parts, patient health monitoring to alert severe health conditions, and credit card fraud detection. Thus unsurprisingly, it has a rich history of research in the machine learning, information theory, data mining, and statistics communities. Techniques such as clustering techniques in data-mining, one-class classifiers using support vector machines or density estimators in machine learning, and naive bayesian networks have all been studied for their application in outlier detection. The appropriate method varies depending on the problem domain, the amount of supervision (labeled data), the amount of apriori modeling, and the dimensionality of the datasets. Please refer to Chandola et. al [24] for a comprehensive overview of the topic, Hodge et. al [48] for a survey of machine learning and statistical approaches, and Markou et. al [78] for a survey of statistical approaches.

Scorpion does not attempt to solve this problem, we assume that the outliers have already been identified and labeled.

7.3.3 RESULT EXPLANATION

Rule-based learning algorithms have long been used to generate human understandable predicates to distinguish positively and negatively labeled datasets. Classification and regression trees [20] are a popular class of learning algorithms that build rules in disjunctive normal form. The *DT* algorithm described in Chapter 4 is based on regression tree learning algorithms. These algorithms can be used in conjunction with outlier detection techniques to identify and describe outliers in datasets.

The main contrast between Scorpion and traditional outlier explanation is that Scorpion's input is not a dataset with individually labeled records. Instead, the records are labeled as groups based on how the data was aggregated in the SQL query. Thus the goal is to differentiate the influential subset of the positively labeled records, whereas traditional outlier explanation tries to describe all of the positively labeled records.

Why Explanation

In the past year, there have been a number of projects that, like Scorpion, explain outliers of aggregation queries. Roy et. al [92] extend this model to support multi-table queries that compare ratios between multiple aggregation queries, and develop a formal approach to

identify and describe a minimal subset of the input data. Their work also describes how to leverage materialized data cubes for simple *COUNT()* queries.

The DBRx [23] system is a general purpose data-cleaning system that identifies and explain errors in result tuples that violate constraints in the form of predicates. In contrast to Scorpion, they support subqueries as well as aggregation and non-aggregation queries. In addition, the system differentiates between explanations that can be generated with access to the result's lineage, and when the lineage is not available. Given the set of result errors, each with weight 1, DBRx traverses the query's operator tree top-down and distributes the weights to the result's operator lineage. A rule-learning algorithm then contructs a disjunctive predicate to cover the input tuples with non-zero weight.

Query Transformation

A number of database projects have tackled the problem of transforming either the input dataset or the user's SQL to cause desired changes in the result set.

Tiresias [81] is a system that allows users to specify incorrect results of a TiQL (constrained version of datalog) query, and will identify changes to the input database that fix the incorrect values. It encodes the problem as input to a Mixed Integer Program solver, which generates a solution. The VCC [79] work by the same authors uses a SAT solver to provide similar functionality to errors as the result of boolean expressions. In both cases, the solutions make tuple-at-a-time modifications to the database, rather than predicate-at-a-time. In addition, these techniques need to encode the the relevant database contents into the MIP or SAT problem, which limits the scalability to tens or low hundreds of tuples.

The *Why Not?* problem [26, 50, 107] seeks to understand why records that should be in the result are not present. Huang et. al [50] and Tiresias [81] explore how to change the database state on a per-tuple basis, whereas an alternative formulation of the problem focuses on changes to the SQL query [26, 107].

We note that these problems are similar to the Query-by-Example problem [125, 126], which attempts to synthesize a query given a database and example result tuples. In contrast, the above problems try to learn a modification to an original query.

General Explanation

Sunita el al. apply statistical approaches to similar applications that explore and explain values in an OLAP data cube. iDiff [96] uses an information-theoretic approach to generate summary tuples that explain why two subcubes' values differ (e.g., higher or lower). Their cube exploration work [97] uses the user's previously seen subcubes during a drill-down session

to estimate the expected values of further drill-down operations. The system recommends the subcube most differing from expectation, which can be viewed as an "explanation". RELAX [99] lets users specify subcube trends (e.g., drop in US sales from 1993 to 1994) and finds the coarsest context that exhibits the similar trend. Scorpion differs by explicitly using influence as the optimization metric, and supports additional information such as hold-out results and error vectors.

MRI [35] is designed in the context of collaborative ratings, and searches for a predicate over the user attributes (e.g., age, state, sex) that most explains average rating of a movie or product (e.g., IMDB ratings). Their work is optimized for the $AVG()$ operator and uses a randomized hill climbing algorithm to find the most influential cuboid in the rating's OLAP lattice.

PerfXplain [66] explains why some MapReduce [37] jobs ran faster or slower than others. The authors provide a query language that lets users easily label pairs of jobs as normal or outliers, and uses a decision tree to construct a predicate that best describes the outlier pairs. This problem is similar to traditional outlier explanation where examples are labeled individually.

Domain Specific Algorithms

Explaining and detecting aggregate outliers has been explored in specialized settings. In network analysis, this is described as the heavy hitters problem. A process consumes a data stream of packet metadata (tuple of source and destination ip) and seeks to find precise descriptions of source and destination subnets that contribute above a pre-specified fraction of the network traffic (the heavy-hitters). This can be viewed as a specialized version of the outlier explanation problem in a streaming scenario for the SQL query:

$$SELECT\ count(*)\ FROM\ network\ GROUP\ BY\ window$$

8 Conclusion

Data-driven decision making and data analysis grown in both importance and availability in the past decade, and has seen increasing acceptance in the broader population. Visual tools that enable non-technical users to explore and make sense of their datasets is challenging, both in terms of developing the systems that can automate the manual and error-prone data analysis tasks and designing intuitive interfaces to these systems

In this thesis, we explored several techniques to help address a common data analysis task thas is ill-served by existing visual analytical tools. Specifically, although visualization tools are well suited to identify patterns in datasets, they do not help users characterize surprising trends or outliers in the visualization and leave that task to the user. In response, we developed the system, algorithms, and interface of an end-to-end visualization tool and found that it can effectively help analysts answer questions about outliers in their data.

Building upon these results, we proposed the design of a general Data Visualization Management System (DVMS) that combines the data processing and optimization features of a database system with the interactive and visualization properties of a visualization system. The integrated design enables a number of powerful visualization features such as those developed in this dissertation, as well as a number of promising end-to-end data visualization optimization techniques.

Bibliography

[1] Benchmark of serial lp solvers. http://plato.asu.edu/ftp/lpfree.html. Accessed: 2014-07-08.

[2] Tableau. http://www.tableausoftware.com.

[3] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom, and Janet Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1997.

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. *EuroSys*, 2013.

[5] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *DMKD*, pages 94–105, 1998.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. on VLDB*, pages 487–499, 1994.

[7] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, pages 1–26, 2014.

[8] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *SSDM*, 2004.

[9] Yael Amsterdamer, Susan Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. In *PVLDB*, 2012.

[10] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram LudÃdscher. Efficient provenance storage over nested data collections. In *EDBT*, 2009.

[11] Douglas Bates and Martin Maechler. *lme4: Linear mixed-effects models using S4 classes*, 2009. R package version 0.999375-31.

[12] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic reduction of query result sets for interactive visualization. *IEEE Big Data Visualization*, 2013.

[13] Richard A. Becker and William S. Cleveland. Brushing scatterplots. *Technometrics*, 1987.

[14] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation, 1976.

[15] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. DataHub: Collaborative Data Science and Dataset Version Management at Scale. *ArXiv e-prints*, September 2014.

[16] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE11*, pages 1151–1162, 2011.

[17] RAJENDRA BOSE and JAMES FREW. Lineage retrieval for scientific data processing: A survey. In *ACM Computing Surveys*, 2005.

[18] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *InfoVis*, 2009.

[19] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *InfoVis*, 2011.

[20] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman & Hall, New York, NY, 1984.

[21] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. Vistrails: Visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 745–747, New York, NY, USA, 2006. ACM.

[22] Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alex Kalinin, Olga Papaemmanouil, and Stan Zdonik. Query steering for interactive data exploration. In *Proceedings of CIDR'13*, 2013.

[23] Anup Chalamalla, Ihab F. Ilyas, Mourad Ouzzani, and Paolo Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD Conference*, pages 445–456, 2014.

[24] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Outlier detection: A survey, 2007.

[25] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *SOSP*, pages 101–114, 2011.

[26] Adriane Chapman and H. V. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 523–534, New York, NY, USA, 2009. ACM.

[27] Adriane P. Chapman, H.V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[28] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *SIGMOD*, 2004.

[29] Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD*, 1998.

[30] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. In *Foundations and Trends in Databases*, 2009.

[31] Jaeyoung Choi, James Demmel, Inderjit S. Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance. In *PARA'95*, pages 95–106, 1995.

[32] William S. Cleveland and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):pp. 531–554, 1984.

[33] Y. Cui, J. Widom, and J. L. Viener. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems*, 1997.

[34] TomaÅ¿ Curk, Janez DemÅąar, Qikai Xu, Gregor Leban, UroÅą Petrovic, Ivan Bratko, Gad Shaulsky, and BlaÅ¿ Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21:396–398, February 2005.

[35] Mahashweta Das, Sihem Amer-Yahia, Gautam Das, and Cong Yu. Mri: Meaningful interpretations of collaborative ratings. In *PVLDB*, volume 4, 2011.

[36] Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. Provenance in scientific workflow systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[38] Mary T. Dzindolet, Scott A. Peterson, Regina A. Pomranky, Linda G. Pierce, and Hall P. Beck. The role of trust in automation reliance. *International Journal of Human-Computer Studies*, 58(6):697 – 718, 2003.

[39] Montserrat Fuentes, Bowei Xi, and William S. Cleveland. Trellis display for modeling data from designed experiments. *Statistical Analysis and Data Mining*, 4(1):133–145, 2011.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[41] J Goecks, A Nekrutenko, J Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. In *Genome Biology*, 2010.

[42] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining Knowledge Discovery*, 1(1):29–53, January 1997.

[43] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *KDD*, 1997.

[44] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[45] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[46] Jeffery Heer and Ben Shneiderman. Interactive dynamics for visual analysis. http://queue.acm.org/detail.cfm?id=2146416.

[47] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 14:1189–1196, 2008.

[48] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[49] David A. Holland, Uri Braun, Diana Maclean, Kiran kumar Muniswamy-reddy, and Margo I. Seltzer. Choosing a data model and query language for provenance, 2008.

[50] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[51] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 2007.

[52] Robert Ikeda. Provenance in data-oriented workflows. 2012.

[53] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.

[54] Robert Ikeda, Semih Salihoglu, and Jennifer Widom. Provenance-based refresh in data-oriented workflows. In *CIKM*, pages 1659–1668, 2011.

[55] Robert Ikeda, Akash Das Sarma, and Jennifer Widom. Logical provenance in data-oriented workflows? In *ICDE*, pages 877–888, 2013.

[56] Robert Ikeda and Jennifer Widom. Panda: A system for provenance and data. In *IEEE Data Engineering Bulletin*, 2010.

[57] Jean-Francois Im, Felix Giguere Villegas, and Michael J. McGuffin. Visreduce: Fast and responsive incremental information visualization of large datasets. In *BigData Conference*, 2013.

[58] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, New York, NY, USA, 2007. ACM.

[59] Z. Ivezi, J.A. Tyson, E. Acosta, R. Allsman, S.F. Anderson, et al. LSST: From science drivers to reference design and anticipated data products.

[60] T.J. Jankun-Kelly, Kwan-Liu Ma, and Michael Gertz. A model and framework for visualization exploration. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):357–369, 2007.

[61] Niranjan Kamat, Prasanth Jayachandran, Kathik Tunga, and Arnab Nandi. Distributed and Interactive Cube Exploration. In *ICDE*, 2014.

[62] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Heer Jeffrey. Enterprise data analysis and visualization: An interview study. *VAST*, 2012.

[63] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Advanced Visual Interfaces*, 2012.

[64] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *VLDB*, 1990.

[65] Alicia Key, Bill Howe, Daniel Perry, and Cecilia R. Aragon. Vizdeck: self-organizing dashboards for visual analytics. *SIGMOD*, 2012.

[66] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Perfxplain: debugging mapreduce job performance. *VLDB*, 5(7):598–609, March 2012.

[67] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *OSDI*, 2010.

[68] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SOSP*, 41(6):321–334, October 2007.

[69] Heidi Kuehn, Arthur Liberzon, Michael Reich, and Jill P. Mesirov. Using genepattern for gene expression analysis. *Curr. Protoc. Bioinform.*, Jun 2008.

[70] John D. Lee and Katrina A. See. Trust in automation: Designing for appropriate reliance. *HUMAN FACTORS*, 46:50–80, 2004.

[71] Lauro Didier Lins, James T. Klosowski, and Carlos Eduardo Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 2013.

[72] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. immens: Real-time visual querying of big data. *EuroVis*, 2013.

[73] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. Scalable lineage capture for debugging disc analytics. In *SOCC*, pages 17:1–17:15, 2013.

[74] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006.

[75] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, November 2009.

[76] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. Show me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[77] Michael V. Mannino, Paicheng Chu, and Thomas Sager. Statistical profile estimation in database systems. *ACM Computer Surveys*, 1988.

[78] Markos Markou and Sameer Singh. Novelty detection: A review - part 1: Statistical approaches. *Signal Processing*, 83:2003, 2003.

[79] Alexandra Meliou, Wolfgang Gatterbauer, Suman Nath, and Dan Suciu. Tracing data errors with view-conditioned causality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 505–516, New York, NY, USA, 2011. ACM.

[80] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Reverse data management. *PVLDB*, 2011.

[81] Alexandra Meliou and Dan Suciu. Tiresias: The database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 337–348, New York, NY, USA, 2012. ACM.

[82] P. Missier, N. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*, 2010.

[83] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6):743–756.

[84] Kiran-Kumar Muniswamy-Reddy, Joseph Barillariy, Uri Braun, David A. Holland, Diana Maclean, Margo Seltzer, and Stephen D. Holland. Layering in provenance-aware storage systems. Technical Report 04-08, Harvard, 2008.

[85] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *NetDB*, 2005.

[86] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. In *Concurrency and Computation: Practice and Experience*, pages 1067–1100, 2006.

[87] Aude Oliva and Antonio Torralba. Building the gist of a scene: the role of global image features in recognition. In *Progress in Brain Research*, 2006.

[88] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[89] Aditya Parameswaran, Neoklis Polyzotis, and Hector Garcia-Molina. Seedb: Visualizing database queries efficiently. *PVLDB*, 2014.

[90] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.

[91] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[92] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *SIGMOD Conference*, pages 1579–1590, 2014.

[93] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, September 2007.

[94] Andrea Saltelli. The critique of modelling and sensitivity analysis in the scientific discourse. an overview of good practices. *TAUC*, October 2006.

[95] Andrea Saltelli, Karen Chan, E Marian Scott, et al. *Sensitivity analysis*, volume 134. Wiley New York, 2000.

[96] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, 1999.

[97] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-driven exploration of olap data cubes. In *EDBT*, 1998.

[98] Sunita Sarawagi and Gayatri Sathe. i3: Intelligent, interactive investigaton of olap data cubes. In *SIGMOD*, 2000.

[99] Gayatri Sathe and Sunita Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, 2001.

[100] Arvind Satyanarayan and Jeffrey Heer. Lyra: An interactive visualization design environment. *EuroVis*, 2014. http://idl.cs.washington.edu/projects/lyra/.

[101] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[102] Chris Stolte and Pat Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. *InfoVis*, 2002.

[103] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, 2009.

[104] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and scidb. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[105] Pablo Tamayo, Yoon-Jae Cho, Aviad Tsherniak, Heidi Greulich, et al. Predicting relapse in patients with medulloblastoma by integrating evidence from clinical and genomic features. *Journal of Clinical Oncology*, page 29:1415âĂŞ1423, 2011.

[106] Jenifer Tidwell. *Designing interfaces*. " O'Reilly Media, Inc.", 2010.

[107] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 15–26, New York, NY, USA, 2010. ACM.

[108] Chris Weaver. Building highly-coordinated visualizations in improvise. In *INFOVIS*, 2004.

[109] Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. An analytic data engine for visualization in tableau. In *SIGMOD*, 2011.

[110] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[111] Hadley Wickham. ggplot2. ggplot2.org.

[112] Hadley Wickham. Bin-summarise-smooth: a framework for visualising large data. Technical report, had.co.nz, 2013.

[113] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford, 2004.

[114] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., 2005.

[115] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. Strategies for crowdsourcing social data analysis. In *CHI*, 2012.

[116] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

[117] Allison Woodruff and Michael Stonebraker. Buffering of intermediate results in dataflow diagrams. In *ISVL*, 1995.

[118] Eugene Wu, Samuel Madden, and Michael Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, 2013.

[119] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *SIGCOMM*, pages 383–394, 2014.

[120] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[121] Peter Zadrozny and Raghu Kodali. *Big Data Analytics Using Splunk*. Apress, Berkeley, CA, 2013.

[122] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[123] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.

[124] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, pages 615–626. ACM, 2010.

[125] Moshé M. Zloof. Query by example. In *American Federation of Information Processing Societies*, pages 431–438, 1975.

[126] Moshe M. Zloof. Query-by-example: A data base language. *IBM systems Journal*, 16(4):324–343, 1977.