Demonstration of SMOKE: A Deep Breath of Data-Intensive Lineage Applications

Fotis Psallidas
Computer Science, Columbia University
fotis@cs.columbia.edu

Eugene Wu
Computer Science, Columbia University
ewu@cs.columbia.edu

ABSTRACT

Data lineage is a fundamental type of information that describes the relationships between input and output data items in a workflow. As such, an immense amount of data-intensive applications with logic over the input-output relationships can be expressed declaratively in lineage terms. Unfortunately, many applications resort to hand-tuned implementations because either lineage systems are not fast enough to meet their requirements or due to no knowledge of the lineage capabilities. Recently, we introduced a set of implementation design principles and associated techniques to optimize lineage-enabled database engines and realized them in our prototype database engine, namely, SMOKE. In this demonstration, we showcase lineage as the building block across a variety of data-intensive applications, including tooltips and details on demand; crossfilter; and data profiling. In addition, we show how SMOKE outperforms alternative lineage systems to meet or improve on existing hand-tuned implementations of these applications.

ACM Reference Format:
Fotis Psallidas and Eugene Wu. 2018. Demonstration of SMOKE: A Deep Breath of Data-Intensive Lineage Applications. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183713.3193537

1 INTRODUCTION

Data lineage describes the relationship between individual input and output data items of a workflow, and is a fundamental type of information for any application that requires an understanding of the input-output derivation process. As such, data lineage can be pertinent across applications of several domains including debugging, data integration, auditing, security, network diagnostics, provisioning, explaining query results, data cleaning, iterative analytics, and interactive visualizations. This ubiquity of lineage across domains highlights the importance of lineage-enabled systems.

The core problem that lineage systems need to address is to capture lineage at the moment of workflow execution with the goal to streamline future queries over lineage. Consider the following:

EXAMPLE 1. *Figure 1 shows two views* V_1 *and* V_2 *generated from queries over a database. Linked brushing is an interaction technique where users select marks (e.g., circles) in one view, and marks derived from the same records are highlighted in other views. This functionality can be expressed with lineage queries (i.e., as a backward query from selected circles in* V_1 *to input records followed by a forward query to highlight bars in* V_2*) and optimized by a lineage system.*

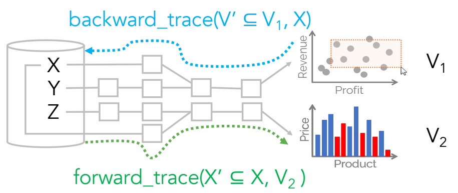


Figure 1: Two workflows generate visualization views V_1 **and** V_2**. Then, a linked brushing interaction highlights in red marks in** V_2 **that share the same input records with selected marks of** V_1**. This interaction can be expressed as a backward query from selected circles in** V_1 **followed by a forward query to highlight the bars in** V_2**.**

Unfortunately, current lineage systems incur either high lineage capture overhead, or high lineage query processing costs, or both. As a result, applications that could be expressed in lineage terms resort to manual implementations. For instance, the linked brushing in our example is typically implemented manually to meet interactive latency requirements (e.g., <150ms). Understanding this discrepancy is both important—manually implementing lineage-based features for increasingly complex applications is error-prone and results in applications that are hard to maintain and optimize—and bewildering—why are application developers forced to manually compile lineage-based logic into physical implementations?

Our recent work [20, 21] showed how four simple design principles enable fast lineage capture and lineage query performance in an in-memory database engine. Our prototype, namely, SMOKE, instruments physical operators to tightly integrate lineage capture logic within operator execution logic and uses write-efficient indexes to store lineage (tight integration principle). Furthermore, SMOKE piggybacks lineage in data structures constructed and reused during normal execution (i.e., hash tables) for faster lineage capture (reuse principle). In addition, if queries that process lineage are known up-front, SMOKE enables optimizations that prune captured lineage (a-priori knowledge principle) and push the logic of such queries into the lineage capture phase (lineage consumption principle).

The goal of our demonstration is to showcase the potential of performant lineage-enabled database systems towards the optimization of data-intensive applications. To this end, we will showcase a variety of popular user-facing applications with challenging performance requirements (i.e., tooltips and details on demand, crossfilter, and interactive data profiling) by first expressing them in lineage terms. Per application, we present head-to-head comparisons with state-of-the-art lineage systems and existing hand-tuned implementations to demonstrate the performance benefits of using SMOKE.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3193537

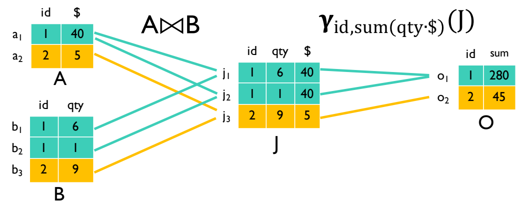


Figure 2: Input, intermediate, and output relations for query $\gamma_{\text{id,sum(qty}\times\$)}(A \bowtie B)$. Lines signify edges of the lineage (or provenance) graph and colored by output record.

2 BACKGROUND AND SMOKE OVERVIEW

To provide background on lineage challenges; lineage systems; Smoke; and lineage applications, we use the example in Figure 2.

Figure 2 illustrates a relational workflow consisting of a natural join between tables A and B followed by grouping on id and aggregating on sum(qty · \$). Lineage represents the dependencies between input and output records for each operator [4], illustrated as lines. The set of lines constitutes the edges in the *lineage (or provenance) graph* of the workflow. Lineage queries recursively follow these pointers from workflow output to source input records (a `backward()` query) or vice versa (a `forward()` query). For instance, the **green** lines show that o_1 depends on a_1 twice and once each for b_1 and b_2; the **orange** lines show that b_3 contributes to o_2 once. The core challenge of a lineage system is to capture (or materialize) a representation of the lineage graph, without slowing down the workflow, in order to answer lineage queries quickly.

State-of-the-art techniques and lineage systems can be classified as lazy or eager. *Lazy* [5, 7] approaches are based on the observation that O.id in the output relation directly corresponds to the id of the input records it depends upon. In other words, in some cases backward lineage queries can be rewritten as relational queries over input relations. For example, `backward(o₁, B)` can be rewritten as $\sigma_{B.id=o_1.id}(B)$. Lazy approaches benefit from not materializing the lineage graph during the workflow execution. However, the rewritten lineage queries are either slow or require physical design options (i.e., indexes and views) to be pre-built. In addition, rewriting in many cases is not possible without additional overheads if the workflow is non-invertible (e.g., if id was not projected in O).

Eager approaches materialize the lineage graph during workflow execution and can be further classified as logical or physical. *Logical* approaches [1, 3, 6, 15] rewrite the workflow into either (a) one that annotates each output record with its lineage information by extending the output schema (e.g., O's schema is extended with the full or partial schemas of A and B) or (b) to multiple queries that generate lineage relations to store tuple-level input-output relationships (e.g., each operator's lines in Figure 2 is a separate relation). Finally, physical approaches re-implement operators to write input-output relationships to an external lineage system [8–10, 23]. Each eager approach answers lineage queries with a different technique depending on the induced data model for the lineage graph.

Although eager approaches typically execute lineage queries faster than lazy approaches, their capture overhead can severely impact workflow execution performance [21], often by orders of magnitude. Logical approaches are affected by the relational lineage graph representation, extra indexing and projection steps, and expensive joins. Physical approaches are affected by per-arrow communication costs with the lineage system, write-inefficient indexes, and the lack of co-optimization of the lineage capture with the query execution. Finally, many applications use the output of lineage queries as inputs to further analyses and queries. For such applications, it is not clear how to take advantage of this apriori knowledge, if available, with either physical or logical approaches.

As a result, while lineage has the potential to express and optimize applications across a vast amount of domains (e.g., from auditing to data cleaning to explaining outliers), the performance overheads of existing approaches deter the direct use of lineage query processing systems in practice. This effect is prevalent in data-intensive interactive applications (e.g., interactive visualizations and profiling) where developers resort to hand-tuned implementations to meet their demanding interactive latency requirements.

To this end, we introduced Smoke [20, 21] as a lineage-enabled query engine that applies four design principles (Section 1) to avoid the shortcomings of alternative approaches and meet the performance requirements of data-intensive lineage applications. Smoke improves upon logical and physical approaches by storing lineage in read- and write-efficient indexes to avoid relational encodings of the lineage graph, extra indexing steps, and inefficient indexes. Furthermore, Smoke introduces a physical algebra that tightly integrates lineage capture and workflow execution to avoid API calls, extra projection steps, and expensive joins. Finally, Smoke leverages knowledge over possible future lineage queries to (a) prune or partition lineage indexes or (b) materialize views during workflow execution with the goal to streamline these lineage queries.

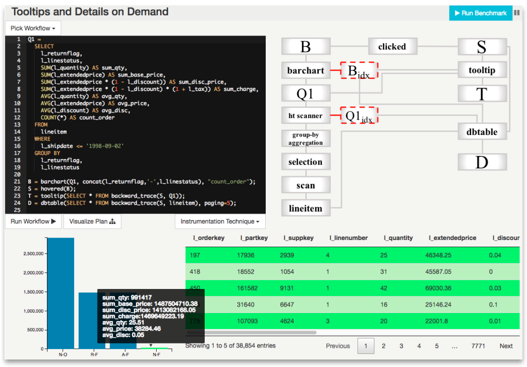
3 DEMONSTRATION

The purpose of our demo is to showcase 1) the variety of applications that fast lineage systems can express and support and 2) that the performance benefits of careful lineage capture and query designs in Smoke outperform alternative lineage systems *and* can be on par with or improve on hand-tuned implementations. To do so, our demonstration includes three interactive applications: tooltips and details-on-demand (Section 3.1), crossfilter (Section 3.2), and interactive data profiling (Section 3.3). Next, along with their description, we discuss how participants can interact with them and how they are live-benchmarked to show performance differences.

3.1 Tooltips and Details-On-Demand

Visualizations often use tooltips to show additional attributes of a visualized output record, and use details-on-demand to render the input records that contributed to a mark in the visualization. For instance, Figure 3 shows our interface for a visualization that executes and renders each output record from TPC-H Q1 as a bar in a barchart; the visualization process is also modeled as a relational workflow [24]. The bar height corresponds to one of Q1's 8 aggregation functions, and users can hover over any bar to see a tooltip with the full output record. Furthermore, users can click on a bar to see the input `lineitem` records that computed the bar's value in the bottom right table. Finally, our interface shows the workflow code and plan to better explain the process to participants.

Techniques. We will let participants use the application using Smoke-based, lazy, and eager approaches as described in Section 2.



```
Q1 =
  SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_quantity) AS sum_qty,
    SUM(l_extendedprice) AS sum_base_price,
    SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    AVG(l_quantity) AS avg_qty,
    AVG(l_extendedprice) AS avg_price,
    AVG(l_discount) AS avg_disc,
    COUNT(*) AS count_order
  FROM
    lineitem
  WHERE
    l_shipdate <= '1998-09-02'
  GROUP BY
    l_returnflag,
    l_linestatus

B = barchart(Q1, concat(l_returnflag,'-',l_linestatus), "count_order");
S = hovered(B);
T = tooltip(SELECT * FROM backward_trace(S, Q1));
D = dbtable(SELECT * FROM backward_trace(S, lineitem), paging=5);
```

```
sum_qty: 991417
sum_base_price: 1487504710.38
sum_disc_price: 1413082168.05
sum_charge:1469649223.19
avg_qty: 25.51
avg_price: 38284.46
avg_disc: 0.05
```

l_orderkey	l_partkey	l_suppkey	l_linenumber	l_quantity	l_extendedprice	l_discour
197	17936	2939	4	25	46348.25	0.04
418	18552	1054	1	31	45587.05	0
450	161582	9131	1	42	69030.36	0.03
[illegible]	31640	6647	1	16	25146.24	0.1
775	107093	4624	3	20	22001.8	0.01

Showing 1 to 5 of 38,854 entries   Previous 1 2 3 4 5 … 7771 Next

Figure 3: Tooltips and details-on-demand interface.

Benchmark. This application supports barchart visualizations for TPC-H queries Q1, Q3, Q10, and Q12. The benchmark compares the alternative lazy and eager techniques (Section 2) with SMOKE in terms of initial visualization load times as well as interaction response times for simulated tooltip and details-on-demand interactions for every bar in each query's bar chart visualization.

3.2 Crossfilter

Crossfilter is an important interaction technique to help explore correlated statistics across multiple visualization views. In the common setup, multiple group-by queries along different attributes are each rendered as, say, bar charts. When the user highlights a bar (or set of bars) in one view, the other views update by considering only the subset that contributed to the highlighted bar(s).

As the initial queries are group-by aggregations, recent research proposes variations of data cubes to accelerate crossfiltering [12, 13, 17]. However, it can take minutes or hours to construct data cubes. Such offline time is not available if the user has loaded a new dataset (e.g., into Tableau) and wants to explore as soon as possible. This has recently been referred to as the cold-start problem for interactive visualizations [2]. To address this problem we can cast a crossfilter interaction as a backward query from highlighted bar(s) to the input dataset followed by computing the group-by queries on the traced input subset (i.e., on the output of the backward query).

Our demonstration aims to show how lineage-based techniques can make use of the lineage capture and query capabilities of SMOKE to address this problem. Participants will interact with the interface in Figure 4, which renders the Ontime [16] dataset using 6 visualization views of group-by `COUNT` aggregations on <state>, <date>, <departure delay>, <carrier>, <year>, and <month>. As a crossfilter example, in Figure 4 we restrict the year range to [2005-2008] and the views are updated by considering only the tuples in this range.
Techniques. We compare four techniques: **LAZY** uses the lazy approach (i.e., rewrites the backward lineage query as a selection over the input relation using a full-table scan) and re-executes the group-by queries on the lineage subset. **BT** uses the indexes of SMOKE to identify the lineage subset, but re-runs the group-by queries (which requires re-building group-by hashtables). **BT+FT** also captures forward lineage indexes and uses them to incrementally update the visualized views and avoid rerunning the group-by queries. Finally, the **DATA CUBE** technique uses our hand-optimized version of existing visualization-optimized cube constructions [12, 13].

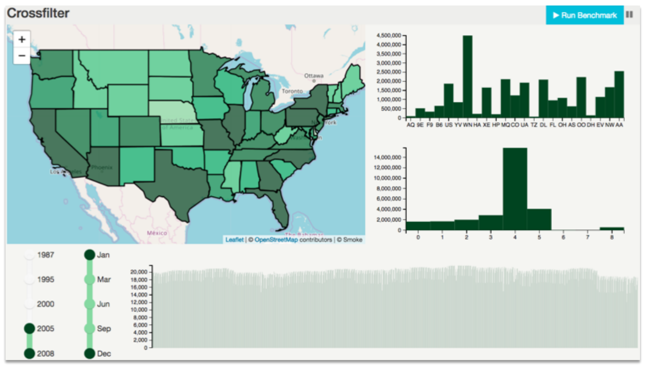


Figure 4: Crossfilter interface.

Benchmark. The benchmark will simulate crossfilter interactions in each of the visualization view and compare the time to render the initial visualization (to measure lineage capture overheads) as well as individual and cumulative interaction response times.

3.3 Interactive Data Profiling

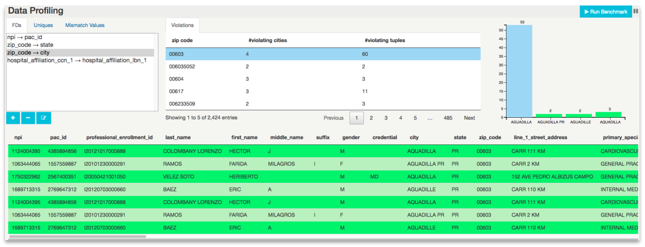
Data profiling studies the statistics and quality of datasets (e.g., constraint checking; data type extraction; or key identification), and interactive data profiling [14] allows users to interactively profile and examine the reasons for these results. Recent systems include extensible data profiling platforms (e.g., METANOME [18]), data wrangling and cleaning tools (e.g., Trifacta Wrangler, Profiler [11]) and user-guided functional dependency miners (e.g., UGUIDE [22]).

In such systems, profiling results can be viewed as the results of data-processing workflows and the interactive profiling functionality corresponds to inspecting raw (or summarized) inputs that contributed to these output statistics. For instance, UGUIDE mines datasets for functional dependencies (FDs) and presents violations of candidate FDs to the user to validate. Similarly, data cleaning applications typically render summary statistics as bar charts or heat maps [11] that the user can interactively inspect.

In this application, we provide an interactive interface for a common set of interactive profiling tasks. These include evaluating common constraints (i.e., functional dependency, uniqueness, and mismatch constraint checks), rendering statistics over corresponding constraint violations, and generating previews of the violating records. These constraints have been shown to be expressible as relational workflows and, as such, the interactive profiling tasks of this application can be expressed in lineage terms.

For instance, checking an FD A⟶B over a table T will output the distinct values a ∈ T.A that violate the FD. It can be expressed as the query Q_{CD} = `SELECT A FROM T GROUP BY A HAVING COUNT(DISTINCT B) > 1`. Interactive data profiling further requires identifying and connecting the input tuples {t ∈ T | t.A = a} with each violating value a. These inputs can then be used to collect statistics over violations or prompt users for cleaning purposes. The key observation is that these inputs are precisely the backward lineage for the output value a of the FD checking query.

Figure 5 shows the interface that participants will use to perform interactive data profiling. On the top, the left panel is used to select functional dependency, uniqueness, or mismatched value constraints to check. Here the user has selected the FD



npi → pac_id
zip_code → state
zip_code → city
hospital_affiliation_ccn_1 → hospital_affiliation_lbn_1

zip code	#violating cities	#violating tuples
00603	4	60
006035052	2	2
00604	3	3
00617	3	11
006233509	2	3

Showing 1 to 5 of 2,424 entries Previous 1 2 3 4 5 … 485 Next

npi	pac_id	professional_enrollment_id	last_name	first_name	middle_name	suffix	gender	credential	city	state	zip_code	line_1_street_address	primary_speci
1124004395	4385894856	I20121017000888	COLOMBANY LORENZO	HECTOR	J		M		AQUADILLA	PR	00603	CARR 111 KM	CARDIOVASCU
1063444065	1557559887	I20101230000291	RAMOS	FARIDA	MILAGROS	I	F		AGUADILLA PR	PR	00603	CARR 2 KM	GENERAL PRA(
1750322962	2567400351	I20050421001050	VELEZ SOTO	HERIBERTO			M	MD	AQUADILLA	PR	00603	152 AVE PEDRO ALBIZUS CAMPO	GENERAL PRA(
1689713315	2769647312	I20120703000660	BAEZ	ERIC	A		M		AGUADILLE	PR	00603	CARR 110 KM	INTERNAL MED
1124004395	4385894856	I20121017000888	COLOMBANY LORENZO	HECTOR	J		M		AQUADILLA	PR	00603	CARR 111 KM	CARDIOVASCU
1063444065	1557559887	I20101230000291	RAMOS	FARIDA	MILAGROS	I	F		AGUADILLA PR	PR	00603	CARR 2 KM	GENERAL PRA(
1689713315	2769647312	I20120703000660	BAEZ	ERIC	A		M		AGUADILLE	PR	00603	CARR 110 KM	INTERNAL MED

Figure 5: Interactive data profiling interface.

zip_code→city. The middle panel renders a summary of the check results in terms of the zipcodes that have more than one city value. Selecting a violation updates the right panel, which shows the distribution of city values for that zipcode, and the bottom panel, which renders a table with the individual records that contributed to the violations. This table is further restricted by selecting a subset of city values in the top right bar chart. Each interaction in the middle and right panels can be expressed using a backward lineage query.

Evaluating functional dependencies. There are two approaches to translating a functional dependency A→B into relational workflows. The first straightforward approach (**CD**) is described above as Q_{CD}, and capturing backward and forward lineage indexes lets the application generate violation statistics and previews. The second approach (**UG**) is based on an optimization in UGuide's Metanome-based implementation. Through correspondence with the authors, it turns out that the implementation effectively simulates lineage indexes, and thus we describe it in lineage terms. We first evaluate $Q_{ug,attr}$=SELECT attr FROM T GROUP BY attr HAVING COUNT(1) > 1 for the attributes in the FD attr$\in \{A, B\}$, and capture lineage. We backward trace each $a \in Q_{ug,A}$ to the input T, and forward trace each lineage record to $Q_{ug,B}$. If more than one distinct b values are in the forward traced output, then the FD is violated, and the lineage indexes connect the violation with the tuples that contributed to the violation. CD is typically faster than UG for the evaluation of individual FDs; UG is faster than CD for batch evaluation of FDs.

Uniqueness and mismatches. To check uniqueness for an attribute attr, we simply execute $Q_{ug,attr}$ from above to identify those that are not unique. The backward lineage for an output record corresponds to the input records that contribute to the uniqueness violation. This also illustrates how lineage from the same query can be shared across data profiling algorithms. Finally, mismatches are expressed with predicates over the input table that should evaluate to true but do not. Such constraints are commonly used to identify NULL values and domain or type violations.

Techniques. For FDs, we compare Smoke using both approaches (CD and UG) with UGuide, which implements the UG approach. This is to show that Smoke can improve the performance of hand-tuned implementations. For uniqueness and mismatch checks we compare Smoke with eager and lazy approaches.

Benchmark. This application uses the Physician [19] dataset (2.2m tuples, 0.6GB) to run the interactive data profiling techniques above. The application is pre-populated with four FDs, four uniqueness tests, and four mismatch constraints. The benchmark will simulate the user clicking on each of the constraints and then clicking each of the output violations (top middle and right panels in Figure 5). We will report the cumulative performance in real time for the alternative techniques against the Smoke-based approaches.

4 CONCLUSIONS

This demonstration highlights the potential of fast lineage-enabled relational systems towards the optimization of data-intensive applications. We described how interactive visualizations and interactive data profiling are expressible as a combination of relational workflows and fine-grained lineage queries. Furthermore, we described the design principles behind our in-memory relational execution engine called Smoke that both captures lineage with low overhead *and* supports lineage queries at interactive speeds. Our demonstration shows that Smoke outperforms alternative lineage systems *and* is on par with or improves on the performance of hand-optimized alternatives. Participants can experience first-hand the power of application specification using declarative lineage statements.

ACKNOWLEDGMENTS

This work is supported by NSF grants #1564049 and #1527765.

REFERENCES

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*.
[2] Leilani Battle, Remco Chang, Jeffrey Heer, and Michael Stonebraker. 2015. Position Statement: The Case for a Visualization Performance Benchmark. In *DSIA*.
[3] Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. 2005. DBNotes: A Post-it System for Relational Databases Based on Provenance. In *SIGMOD*.
[4] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. Explaining Outputs in Modern Data Analytics. *PVLDB* 9, 12 (2016), 1137–1148.
[5] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *TODS* 25, 2 (2000), 179–227.
[6] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*.
[7] Robert Ikeda. 2012. *Provenance In Data-Oriented Workflows*. Ph.D. Dissertation. Stanford University.
[8] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for generalized map and reduce workflows. In *CIDR*.
[9] Robert Ikeda and Jennifer Widom. 2010. Panda: A System for Provenance and Data. *Data Engineering Bulletin* 33, 3 (2010), 42–49.
[10] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *PVLDB* 9, 3 (2015), 216–227.
[11] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2012. Profiler: Integrated Statistical Analysis and Visualization for Data Quality Assessment. In *AVI*.
[12] Lauro Lins, James T Klosowski, and Carlos Scheidegger. 2013. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *EuroVis* (2013).
[13] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. 2013. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum (Proc. EuroVis)* 32. Issue 3.
[14] Felix Naumann. 2014. Data Profiling Revisited. *SIGMOD Rec.* (2014).
[15] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. 2017. Provenance-aware Query Optimization. In *ICDE*.
[16] Ontime. Airline On-Time Performance. http://bit.ly/ontime09.
[17] Cıcero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. 2017. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. *TVCG* 23, 1 (2017), 671–680.
[18] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data Profiling with Metanome. *PVLDB* 8, 12 (2015), 1860–1863.
[19] Physicians. Physician Compare National. http://bit.ly/pcomp.
[20] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *CoRR* abs/1801.07237 (2018).
[21] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fined-Grained Lineage Capture At Interactive Speed. *PVLDB* 11, 6 (2018), 719–732.
[22] Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, and Nan Tang. 2017. UGuide: User-Guided Discovery of FD-Detectable Errors. In *SIGMOD*.
[23] Eugene Wu, Samuel Madden, and Michael Stonebraker. 2013. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*.
[24] Eugene Wu, Fotis Psallidas, Zhengjie Miao, Haoci Zhang, Laura Rettig, Yifan Wu, and Thibault Sellam. 2017. Combining Design and Performance in a Data Visualization Management System. In *CIDR*.