SMOKE: Fine-grained Lineage at Interactive Speed

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

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

ABSTRACT

Data lineage describes the relationship between individual input and output data items of a workflow and is an integral ingredient for both traditional (e.g., debugging or auditing) and emergent (e.g., explanations or cleaning) applications. The core, long-standing problem that lineage systems need to address—and the main focus of this paper—is to quickly capture lineage across a workflow in order to speed up future queries over lineage. Current lineage systems, however, either incur high lineage capture overheads, high lineage query processing costs, or both. In response, developers resort to manual implementations of applications that, in principal, can be expressed and optimized in lineage terms. This paper describes SMOKE, an in-memory database engine that provides both fast lineage capture and lineage query processing. To do so, SMOKE tightly integrates the lineage capture logic into physical database operators; stores lineage in efficient lineage representations; and employs optimizations if future lineage queries are known up-front. Our experiments on microbenchmarks and realistic workloads show that SMOKE reduces the lineage capture overhead and lineage query costs by multiple orders of magnitude as compared to state-of-the-art alternatives. On real-world applications, we show that SMOKE meets the latency requirements of interactive visualizations (e.g., < 150ms) and outperforms hand-written implementations of data profiling primitives.

PVLDB Reference Format:
Fotis Psallidas and Eugene Wu. SMOKE: Fine-grained Lineage at Interactive Speed. *PVLDB*, 11(6): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3184470.3184475

1. INTRODUCTION

Data lineage describes the relationship between individual input and output data items of a computation. For instance, given an erroneous result record of a workflow, it is helpful to retrieve the intermediate or base records to investigate for errors. Similarly, identifying output records that were affected by corrupted input records can help prevent erroneous conclusions. These operations are expressed as lineage queries over the workflow: backward queries return the subset of input records that contributed to a given subset of output records while forward queries return the subset of output records that depend on a given subset of input records.

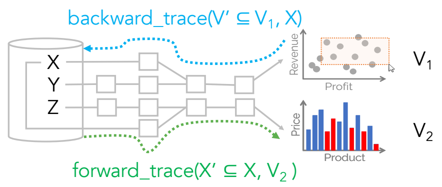

Figure 1: Two workflows generate visualizations V_1 and V_2. A linked brushing interaction highlights in red bars in V_2 that share the same input records with selected circles of V_1. Logically, it is expressed as a backward query from selected circles in V_1 to input tuples followed by a forward query to V_2 to highlight bars.

Any workflow-based application that relies on logic over the input-output relationships can be expressed in lineage terms. As such, lineage is (or can be) integral across many domains, including debugging [12, 31, 34, 40, 59]; data integration [15]; auditing [19]; security [10, 34]; explaining query results [16, 53, 57, 58]; cleaning [8, 25]; and interactive visualizations [60]. This ubiquity highlights the importance of lineage-enabled systems for both traditional as well as emergent domains. To illustrate, consider the interactive visualization in Figure 1:

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 a set of marks in one view and marks derived from the same records are highlighted in the other views. This functionality is typically implemented imperatively in ad-hoc ways [60]. However, it can be expressed declaratively as lineage queries (i.e., as a backward query from selected circles in* V_1 *to input records, followed by a forward query to highlight corresponding bars in* V_2*) and optimized as such by lineage-enabled systems.*

Lineage-enabled systems answer lineage queries by automatically capturing record-level relationships throughout a workflow. A naïve approach materializes pointers between input and output records for each operator during workflow execution and follows these pointers to answer lineage queries. Existing systems primarily differ based on when the relationships are materialized (e.g., *eagerly* during workflow execution or *lazily* reconstructed when executing a lineage query), and how they are represented (e.g., tuple annotations [2, 5, 22, 29] or explicit pointers [40, 59]). Each design trades off between the time and storage overhead to capture

Proceedings of the VLDB Endowment, Vol. 11, No. 6
Copyright 2018 VLDB Endowment 2150-8097/18/02... $ 10.00.
DOI: https://doi.org/10.14778/3184470.3184475

lineage, and lineage query performance. For instance, an engine may augment each operator to materialize a hash index that maps output to input records in order to speed up backward lineage queries. However, the index construction costs can dwarf the operator execution cost by 100× or more [59]—particularly if the operator is highly performant.

As data processing becomes faster, a crucial question—and the main focus of this paper—is whether it is possible to have both negligible lineage capture overhead *and* fast lineage query execution. Unfortunately, current lineage systems incur either high lineage capture overhead, or high lineage query processing costs, or both. Not satisfying these requirements, however, leads developers to abandon declarativity and manually implement lineage-related logic for many data-intensive applications, such as the one in our example.

To this end, Smoke is a fast lineage-enabled in-memory query engine designed to address the major performance overheads in current lineage systems. We designed Smoke based on the careful combination of four design principles that we believe are helpful when incorporating lineage into fast, data-intensive workflow systems:

P1. Tight integration. In high throughput query processing systems, per-tuple overheads incurred within a tight loop—even a single virtual function to store lineage on a separate lineage subsystem [31, 40, 59]—can slow down operator execution by more than an order of magnitude. In response, Smoke introduces a new physical algebra that tightly integrates the lineage capture logic into query execution. In addition, Smoke stores lineage in write-efficient data structures to further reduce the lineage capture overheads.

P2. Apriori knowledge. Lineage applications such as debugging need to capture lineage to answer ad-hoc lineage queries that can trace back or forth to any input, intermediate, or output table. For applications such as interactive visualizations or profiling, however, lineage queries may be known up-front. Smoke uses this apriori knowledge to avoid materializing lineage that will not be queried in the future.

P3. Lineage consumption. Lineage applications rarely require all results of a lineage query (e.g., all records that contributed to an aggregation result) unless the results have low cardinality. Instead, the results are filtered, transformed, and aggregated by additional SQL queries. We term these queries *lineage consuming queries*. If such queries are known up-front, as is typically the case for applications with templated analysis capabilities (e.g., Tableau or Power BI), Smoke pushes physical design optimizations into the lineage capture phase. These optimizations are used to speed up future lineage consuming queries, and can include lineage index partitioning, materializing aggregates, or collecting statistics.

P4. Reuse. Lineage capture introduces significant overhead during query execution due to generating and storing unnecessary amounts of lineage data (e.g., expensive annotations, denormalized forms of lineage). Following the concept of reusing data structures [18], Smoke augments and reuses data structures (i.e., hash tables) constructed during normal query execution to overlap capture and execution costs.

Our contributions are as follows:

- To reduce capture lineage overhead, we introduce a physical algebra that tightly integrates the lineage capture logic within the processing of single and multi-operator plans, and stores lineage in write-efficient indexes. Operators serve the dual purpose of executing the query logic and generating lineage. (Section 3)
- To account for applications with known future lineage query logic, we develop simple yet effective optimizations that use knowledge of future lineage consuming queries to augment, partition, or prune the captured lineage in order to streamline future lineage consuming queries. (Section 4)
- We find that Smoke reduces lineage capture overhead *and* lineage query processing costs by up to multiple orders of magnitude compared to state-of-the-art approaches. Our real-world experiments suggest that Smoke can both express recent interactive visualization and data profiling applications declaratively using lineage constructs, and optimize their performance to be on par with, or faster than hand-written implementations. (Sections 5 and 6)

2. BACKGROUND

We now provide background on lineage capture, an overview of Smoke, and a discussion on lineage applications.

2.1 Fine-Grained Lineage Capture

Our lineage semantics adhere to the transformation provenance semantics of [13, 22, 28] over relational queries.

Base queries. Formally, let the *base query* $Q_{\mp}(D) = O$ be a relational query over a database of relations $D = \{R_1, \cdots, R_n\}$ that generates an output relation O. An application can initially execute multiple base queries $\mathbb{Q}_{\mp} = \{Q_{\mp 1}, \cdots, Q_{\mp m}\}$. For instance, $\mathbb{Q}_{\mp}$ in Figure 1 consists of two base queries that generate the two visualization views.

Lineage and lineage consuming queries. After a base query runs, the user may issue a backward lineage query $L_b(O', R_i)$ that traces from a subset of an output relation $O' \subseteq O$ to a base table R_i, or a forward lineage query $L_f(R_i', O)$ that traces from a subset of an input relation $R' \subseteq R_i$ to the query's output relation O. A lineage query $L(\bullet)$ results in a relation that can be used in another query $C(D \cup \{L(\bullet)\})$ which we term a *lineage consuming query*; a lineage query is a special case of lineage consuming queries: C=`SELECT * FROM` $L(\bullet)$. Finally, C itself can be used as a base query, meaning that another lineage consuming query C' can use C as a base query.[1] We illustrate these definitions with the interactive linked brushing example of Figure 1:

Example 2. *Let* $Q_{\mp 1}(\{X, Y\}) = V_1$ *and* $Q_{\mp 2}(\{X, Z\}) = V_2$ *be the base queries in Figure 1. The linked brushing interaction is expressed as a backward query* $L_b(V_1', X)$ *from the selected circles* $V_1' \subseteq V_1$ *back to the input records in* X *that generated them. The forward lineage query* $F = L_f(L_b(V_1', X), V_2)$ *retrieves the linked bars in* V_2*. A lineage consuming query* $C(D \cup F)$ *can then be used to change the color of the bars to* red*, similar to [60]. Since interactions are expressed in lineage terms, optimizing lineage constructs essentially corresponds to optimizing such applications.*

Lazy and eager lineage query evaluation. How can we answer lineage queries quickly? *Lazy* approaches rewrite lineage queries as relational queries over the input relations—the base queries do not incur capture overhead at the cost of potentially slower lineage query processing [11, 15, 28]. In contrast, we might *Eagerly* materialize data structures during base query execution to speed up future lineage queries [11, 28]. We refer to this problem as lineage capture, and we seek to reduce the capture overhead on the base query execution.

[1]The query model of Smoke includes multi-backward and multi-forward queries as well as refresh and forward propagation queries [28]. We limit the discussion to L_b, L_f, and C in that they form the basis to express more general query constructs.

Lineage capture overview. The eager approach incurs overhead to capture the base query's *lineage graph.* Logically, each edge $a \overset{op}{\longleftrightarrow} b$ maps an operator op's input record a to op's output record b that is derived from a. Backward lineage connects tuples in the query output $o \in O$ with tuples in each input base relation $r \in R_i$ by identifying all end-to-end edges $o \rightsquigarrow r$ for which a path exists between the two records. Forward lineage reverses these arrows. Materializing such end-to-end forward and backward *lineage indexes* can essentially help us streamline lineage consuming queries.

We will present techniques that efficiently capture lineage in a *workload-agnostic* setting by carefully instrumenting operator implementations and in a *workload-aware* setting by tailoring the indexes for future lineage consuming queries if they are known up-front. Next, we review alternative lineage capture techniques that we classify as logical and physical.

Logical lineage capture. This class of approaches stays within the relational model by rewriting the base query into $Q'_{\mp}(\{R_1, \cdots, R_n\}) = O'$, so that its output is annotated with additional attributes of input tuples. Some systems [2, 12] generate a normalized representation of the lineage graph such that a join query between O' and each base relation R_i can create the lineage edges between O' and R_i. The correct output relation O can be retrieved by projecting away the annotation attributes from O'. Alternative approaches [12, 22] output a single denormalized representation that extends O' with attributes of the input relations. Recent work has shown that the latter rewrite rules (Perm [22]) and optimizations leveraging the database optimizer (GProm [44]) incurs lower capture overheads than the former normalized approach.

Although these approaches can run on any relational database and benefit from the database optimizer, they suffer from several performance drawbacks. The normalized representation requires expensive independent joins when running lineage queries. The denormalized representation can incur significant data duplication (e.g., an aggregation output o computed over k input records will be duplicated $k\times$) and require further projections to derive O from O'. Furthermore, indexes are needed to speed up lineage queries.

Physical lineage capture. This class of approaches instruments physical operators to write lineage edges to a lineage subsystem through an API provided by the subsystem; the subsystem stores and indexes the edges, and answers lineage queries [29, 30, 31, 40, 59]. This approach can support black-box operators and decouples lineage capture from its physical representation. However, we found that virtual function calls alone (ignoring cross-process overheads) can slow down data-intensive operators by up to $2\times$. Furthermore, lineage capture with lineage subsystems is not amenable to co-optimization opportunities with the base query execution.

2.2 Approach of Smoke

To this end, we introduce Smoke, an in-memory database engine that avoids the drawbacks of logical and physical approaches. Smoke improves upon logical approaches by physically representing the lineage edges as read- and write-efficient indexes instead of relationally-encoded annotations. We improve upon physical approaches by introducing a physical algebra that tightly integrates lineage capture and relational operator logic to avoid API calls and in a way amenable to co-optimization. Finally, Smoke leverages knowledge of future lineage consuming queries to prune or partition lineage indexes and materialize views to benefit such queries.

The primary focus of this work is to explore mechanisms to instrument physical operator plans with lineage capture logic. To do so, we have built Smoke as a query compilation execution engine using the produce-consumer model [43]. It takes as input the base query $Q_{\mp}$ and an optional workload of lineage consuming queries W; parses and optimizes $Q_{\mp}$ to generate a physical query plan; instruments the plan to directly generate lineage indexes; and compiles the instrumented plan into machine code that generates $Q_{\mp}(D)$ and lineage. Internally, Smoke uses a single-threaded, row-oriented execution model, and leverages hash-based operator implementations that are widely used in fast query engines and are amenable to low-overhead lineage capture. Lineage-aware query optimization and incorporating advanced features (e.g., compression or vectorization) are interesting future steps.

2.3 Lineage Applications

Many applications logically rely on lineage, including but not limited to: debugging [12, 31, 34, 40, 59], diagnostics [56], data integration [15], security [10, 34], auditing [19] (the recent EU GDP regulation [19] mandates tracking lineage), data cleaning [8, 25], explaining query results [16, 53, 57, 58], debugging machine learning pipelines [36, 61], iterative analytics [13], and interactive visualizations [60].

Unfortunately, there is a disconnect between modeling applications in terms of lineage and the performance of existing lineage capture mechanisms. The overheads are enough that application developers resort to manual implementations. For this reason, we center the paper around interactive visualizations. It is a domain that can directly translate to lineage [27, 60]. Yet, it is dominated by hand-written implementations. Furthermore, it imposes strict latency requirements on lineage capture (to show the initial visualizations) and lineage consuming queries (to respond to user interactions). Finally, our experiments on interactive visualizations and data profiling (Section 6.5) seek to argue that lineage does not only provide an elegant logical description of many applications, but it can optimize applications to be on a par with or even *faster* than hand-tuned implementations.

3. FAST LINEAGE CAPTURE

This section describes lineage capture without knowledge of a future workload. First, we present write- and read-efficient lineage index representations (Section 3.1) to map output-to-input or input-to-output record ids (*rids*). Then, we introduce our physical algebra that tightly integrates the lineage capture logic within the base query execution. To do so, we will describe how to instrument single-(Section 3.2) and multi-(Section 3.3) operator plans for fast lineage capture.

3.1 Lineage Representations

Smoke uses two main rid-based lineage representations. Figure 2 below illustrates input and output relations R and O, respectively, and the two rid-based lineage representations for 1-to-N and 1-to-1 relationships between output and input records. We index rids because the indexes are cheap to write (for fast lineage capture) and lookups, that simply index into relations, are fast (for fast lineage query processing). In contrast, indexing full tuples incurs high write costs while indexing primary keys is not beneficial if keys are wide. Furthermore, in-memory engines [1, 20] already create rid lists, as part of query processing, that resemble our indexes and could be reused for the optimization of lineage capture.

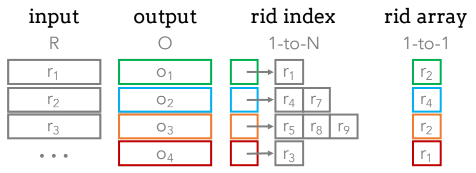

Figure 2: Lineage index representations: rid index for 1-to-N (e.g., γ backward lineage) and rid array for 1-to-1 (e.g., σ) relationships.

Rid Index. 1-to-N relationships are represented as inverted indexes. Consider the backward lineage of `GROUPBY`. The index's ith entry corresponds to the ith output group, and points to an rid array containing rids of the input records that belong to the group. The rid index can also be used for 1-to-N forward lineage relationships, such as for the `JOIN` operator. Following high-performance libraries [21], the index and rid arrays are initialized to 10 elements and grow by a factor of 1.5× on overflow. Our experiments show that array resizing dominates lineage capture costs. Available statistics, however, that allow SMOKE to allocate appropriately sized arrays can reduce lineage capture overheads by up to 60%.
Rid Array. 1-to-1 relationships between output and input records are represented as a single array. Each entry is an rid rather than a pointer to an rid array as in rid indexes.

3.2 Single Operator Instrumentation

Having presented the main lineage index representations, in this section we introduce instrumentation techniques to generate lineage indexes when executing individual relational operators. (Section 3.3 extends support to multi-operator plans.) Our techniques are based on two paradigms: DEFER defers portions of the lineage capture until after operator execution while INJECT incurs the full cost during the base query execution. DEFER is preferable when the overhead on the base query execution *must* be minimized or when it is possible to collect cardinality statistics during base query execution to avoid resizing costs. In contrast, INJECT typically incurs lower overall overhead, but the client needs to wait longer to retrieve the base query results.

Next, we describe both paradigms for core relational operators. Our discussion also illustrates how both paradigms embody the tight integration and reuse principles (**P1** and **P4** from the Introduction). Our focus is on the mechanisms while Section 7 discusses future work to choose between the two paradigms. Full details, code snippets, and additional operators ($\cup$, $\cap$, $-$, $/$, $\times$, $\bowtie_\vartheta$) are in our technical report [50].

3.2.1 *Projection*

Projection under bag semantics does not need lineage capture because the input and output orders and cardinalities are identical. More specifically, the rid of an output (input) record is its backward (forward) lineage. Projection with set semantics is implemented using grouping and we use the same mechanism as that for group-by aggregation (Section 3.2.3).

3.2.2 *Selection*

Selection is an `if` condition in a `for` loop over the input relation, and emits a record if the predicate evaluates to true [42]. Both forward and backward lineage use rid arrays; the forward rid array can be preallocated based on the cardinality of the input relation. INJECT adds two counters, ctr_i and ctr_o, to track the rids of the current input and output records, respectively. If a record is emitted, we set the $\text{ctr}_\text{i}^{\text{th}}$

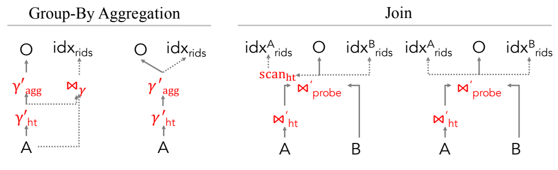

Figure 3: INJECT and DEFER plans for group-by aggregation and join. Dotted arrows are only necessary for lineage capture.

element of the forward rid array to ctr_o, and append ctr_i to the backward rid array. Selectivity estimates can be used to preallocate the backward rid array and avoid reallocations during the append operation. DEFER is equivalent to scheduling INJECT later and requires re-scanning the input.

3.2.3 *Group-By Aggregation*

Query compilers decompose group-by aggregations into two physical operators: γ_ht builds the hash table that maps group-by values to their group's intermediate aggregation state; γ_agg scans the hash table, finalizes aggregation results for each group, and emits output records. Figure 3 shows the plans for both instrumentation paradigms; lineage indexes consist of a forward rid array and a backward rid index.
DEFER. Consider the DEFER plan in Figure 3.a. γ'_ht for DEFER extends γ_ht to store an oid number to each group's intermediate aggregation state. When γ'_agg scans the hash table to construct the output records, it uses a counter to track the output record's rid and assign it to the group's oid value (i.e., oid tracks the output rid of the group in the result). SMOKE then pins the hash table in memory. At a later time, $\bowtie_\gamma$ can scan each record in A, reuse the hash table to probe and retrieve the associated group's oid, and populate the backward rid index and forward rid array.

Although DEFER must scan A twice, the operator's input and output cardinalities can avoid resizing costs during $\bowtie_\gamma$. Also, $\bowtie_\gamma$ can be freely scheduled (e.g., immediately after γ'_ht or during user think time when system resources are free).
INJECT. Consider the INJECT plan in Figure 3.b. γ'_ht this time augments each group's intermediate state with an rid array, say, i_rids, which contains the rids of the group's input records (i.e., its backward lineage). γ'_agg tracks the current output record id oid to set the pointer in the backward index to the bucket's rid array and the values in the forward rid array. Since γ'_agg knows the input and output cardinalities, it can correctly allocate arrays for the backward and forward indexes. The primary overhead is due to reallocations of i_rids during the build phase γ'_ht. As an optimization, our experiments will show that knowing group cardinalities can decrease the lineage capture overhead by up to 60%.

3.2.4 *Join*

SMOKE instruments hash joins in a similar way to hash aggregations. A hash join is split into two physical operators: $\bowtie_\text{ht}$ builds the hash table on the left relation A and $\bowtie_\text{probe}$ uses each record of the right relation B to probe the hash table. Next, we introduce INJECT and DEFER techniques for lineage capture on M:N joins and further optimizations mainly targeting primary key-foreign key joins. For M:N joins, each input record can contribute to multiple output records while each output record is generated from one record of each relation. Hence, SMOKE generates one backward rid array and one forward rid index per input relation.

Inject. Consider the Inject plan for joins in Figure 3.d. The build phase $\bowtie'_{\text{ht}}$ augments each hash table entry with an rid array i_{rids} that contains the input rids from A for that entry's join key. The probe phase $\bowtie'_{\text{probe}}$ tracks the rid for each output record and populates the forward and backward indexes as expected. Note that output cardinalities are not yet known within the $\bowtie'_{\text{probe}}$ phase and we cannot preallocate our lineage indexes. As a result, although the backward rid arrays are cheap to resize, forward rid indexes can potentially trigger multiple reallocations (i.e., if an input record has many matches) which penalize the capture performance.

Defer. Our main observation is that exact cardinalities needed to preallocate the forward rid indexes are known *after* the probe phase and can be used by Defer. To this end, Defer partially defers index construction for the left input relation A (see Figure 3.c). The build phase adds a second rid array, say, o_{rids}, to the hash table entry, in addition to i_{rids} from Inject. When B is scanned during the probe phase, its output records are emitted contiguously, thus o_{rids} need only store the rid of the first output record for each match with a B record. After the $\bowtie'_{\text{probe}}$ phase, the forward and backward indexes for the left relation A can then be preallocated and populated in a final scan of the hash table (scan_{ht} in Figure 3.c). Deferring for B is also possible. However, the benefits are minimal because we need to partition the output records for each hash table entry by the B records that it matches, which we found to be costly.

Further optimizations. If the hash table is constructed on a unique key, then the i_{rids} do not need to be arrays and can be replaced with a single integer. Also, if the join is a primary-key foreign-key join, the forward index of the foreign-key table is an rid array. This is because each record of the foreign-key table contributes to exactly one output record. Furthermore, the output cardinality is the same with the foreign-key table cardinality and we preallocate the backward rid array. Finally, join selectivities can help preallocate forward rid indexes, similarly to how group cardinalities help preallocate backward rid indexes for group-by aggregations.

3.3 Multi-Operator Instrumentation

The naïve way to support multi-operator plans is to individually instrument each operator to generate its lineage indexes. Lineage queries can then use the indexes to trace backward or forward through the plan. This approach is correct and can support any DAG workflow composed of our physical operators. However, it unnecessarily materializes all intermediate lineage indexes even though only the lineage between output and input records are strictly required.

We address this issue with a technique that 1) propagates lineage information throughout plan execution so that only a single set of lineage indexes connecting input and final output relations are emitted and 2) reduces the number of lineage index materialization points in the query plan.

To propagate lineage throughout plan execution, consider a two-operator plan $\text{op}_{\text{p}}(\text{op}_{\text{c}}(\text{R})) = \text{O}$ with input relation R. When op_{p} runs, it will use op_{c}'s backward lineage index to populate its own lineage index with rids that point to R rather than the intermediate relation $\text{op}_{\text{c}}(\text{R})$; op_{c}'s lineage indexes can be garbage collected when not further needed.

To reduce lineage index materialization points, recall that database engines pipeline operators to reduce intermediate results by merging multiple operators into a single pipeline [42]. Operators such as building hash tables are *pipeline breakers* because the input needs to be fully read before the parent operator can run. Within a pipeline, there is no need for lineage capture, but pipeline breakers need to generate lineage along with the intermediate result. In Section 3.2, we showed how pipeline breakers (e.g., hash table construction for the left-side of joins and group-by aggregations) can augment the hash tables with lineage. Parent pipelines that use the same hash-tables for query evaluation (e.g., cascading joins) can also use the lineage indexes embedded in the hash tables to implement the lineage propagation technique above.

Implementation Details. Our engine supports naïve instrumentation for arbitrary relational DAG workflows, and we focused our optimizations for SPJA query blocks composed of pk-fk joins. This was to simplify our engineering and because fast capture for SPJA blocks can be extended to nested blocks by using the propagation technique above. Optimizations for lineage capture across SPJA blocks is interesting future work. We focus on pk-fk joins due to their prevalence in benchmarks and real-world applications and because Inject and Defer for pk-fk joins are identical due to our optimizations (Section 3.2). Thus, the main distinction between Inject and Defer for SPJA blocks is how the final aggregation operator in the block is instrumented—the joins are instrumented identically, while select and project are pipelined. Further details are in our technical report [50].

4. WORKLOAD-AWARE OPTIMIZATIONS

Lineage applications, such as interactive visualizations, often support a pre-defined set of interactions (e.g., filter, pan, tooltip, or cross-filter [14]) that amount to a pre-declared lineage consuming query workload W. This section describes simple but effective optimizations that exploit knowledge of W to avoid capturing lineage (**P2** principle) and generating lineage representations that directly speed up queries in W (**P3** principle). To simplify the discussion, we present each optimization with lineage consuming queries over the base query $\text{Q}_{\doteqdot} = \sigma_{o\_\ orderdate > \text{'2017-08-01'}}(\mathit{orders} \bowtie \mathit{lineitem})$.

4.1 Instrumentation Pruning

Instrumentation pruning disables lineage capture for lineage indexes that will not be used in W. We present two types of pruning that disable lineage capture for specific input relations and lineage directions (i.e., backward or forward).

Pruning input relations. A simple visualization of $\text{Q}_{\doteqdot}$ shows a tooltip of `lineitem` information when a user hovers over a bar. This is expressed as a backward query from an output of $\text{Q}_{\doteqdot}$ to `lineitem`. Thus, we can avoid capturing lineage for `orders` in $\text{Q}_{\doteqdot}$. In general, Smoke does not capture lineage for relations not referenced in the workload W.

Pruning lineage directions. Extending the previous example, it is clear that W will only execute a backward lineage query to `lineitem` and not vice versa. Thus, Smoke can also avoid generating the forward lineage index from `lineitem` to the base query output. The lineage indexes that can be pruned are evident from the lineage consuming queries in W.

4.2 Push-Down Optimizations

User-facing applications rarely present a large set of query results to a user. Instead, they *reduce* the result cardinality with further filter, transformation, and aggregation operations. These reductions are expressed as lineage consuming queries and can be pushed into the lineage capture logic. We present three simple push-down optimizations for fixed and parameterized predicates as well as group-by aggregations. We also provide a brief discussion on the relationship between our optimizations and common provenance semantics.

Selection push-down. Visualizations often update metrics that summarize data based on user selections. For instance, the following query retrieves Christmas shipment order information for parts of the visualization that the user interacts with: $C = \sigma_{\text{shipdate}=\text{`xmas'}}(L_B(O' \subseteq Q_{\mp}(D), \text{orders}))$. Our selection push-down optimization pushes down the predicate `shipdate='xmas'` into lineage capture, so that SMOKE will first check whether the input tuple satisfies the predicate before adding it to the lineage indexes. If the predicate is on a group-by key, SMOKE does not capture lineage for all other groups. This reduces lineage space overheads and usually reduces capture overheads. If the predicate is expensive to evaluate (e.g., slow UDF), it can increase capture overheads.

Data skipping using lineage. Pushing down selections requires fixed predicates. However, interactive visualizations also use parameterized predicates. For instance, a user may use a slider to dynamically change the shipping date (`:p1`): $C = \sigma_{shipdate=:\text{p1}}(L_B(O' \subseteq Q_{\mp}(D), \text{orders}))$. This pattern is ubiquitous in interactive visualizations and applies to faceted search, cross-filtering, zooming, and panning. SMOKE pushes this down by partitioning the rid arrays (standalone, or part of rid indexes) by the predicate attribute. For the example above, SMOKE would partition the rid arrays in the backward index for `orders` by `shipdate`, so that C only reads the rid partition matching the parameter `:p1`. This technique applies to categorical as well as discretized continuous attributes. This makes it attractive for interactive visualizations since outputs are ultimately discretized at pixel granularity [33].

Group-by push-down. Interactions, such as cross-filtering, let users select marks in one view, trace those marks to the input records that generated them, and recompute the aggregation queries in other views based on the selected subset of input records. This pattern is precisely an aggregation query over the backward lineage of the user's selection. SMOKE pushes the group-by aggregation into lineage capture by partitioning the rid arrays on the group-by attributes, and incrementally computing the intermediate aggregation state. This works if the base and lineage consuming query primarily differ in terms of added grouping attributes, and effectively generates data cubes to answer the linage consuming aggregation queries. In contrast to building data cubes offline, which requires separate scans of the database, this optimization piggy-backs on top of the base query's table scans. As with prior work [23, 26, 39], this optimization supports algebraic and distributive functions (e.g., `SUM`, `COUNT`, and `AVG`). To illustrate its importance, we evaluate it extensively in synthetic (Section 6.4) and real-world settings (Section 6.5.1).

Relationships with Provenance Semantics. Popular provenance semantics (e.g., which- [15,24] and why- [6] provenance) can be expressed in SMOKE using lineage consuming queries and pushed down through our optimizations. In other words, SMOKE can operate under alternative provenance semantics depending on the given W. We refer interested readers to further discussion in our technical report [50].

Choosing Optimizations. SMOKE provides applications with different ways to optimize their lineage logic. This poses interesting questions for future work. What SQL extensions (e.g., `CREATE BACKWARD INDEX ON SELECT...`) can provide users with control over the lineage capture? What cost models are needed to choose among capture and optimization options?

Table 1: Lineage capture techniques used in our evaluation.

Abbreviation	Description
Smoke	
BASELINE	SMOKE without lineage capture
SMOKE-D	SMOKE with defer lineage capture
SMOKE-I	SMOKE with inject lineage capture
Logical	
LOGIC-RID	Rid-based annotation
LOGIC-TUP	Tuple-based annotation
LOGIC-IDX	Indexing input-output relations
Physical	
PHYS-MEM	Virtual emit function calls and no reuse
PHYS-BDB	Lineage capture using BerkeleyDB

5. EXPERIMENTAL SETTINGS

Our experiments seek to show that SMOKE (1) incurs significantly lower lineage capture overhead than logical and physical lineage capture approaches, (2) can execute lineage queries faster than lazy, logical, and physical lineage query approaches, and (3) can leverage lineage indexes and workload-aware optimizations to speed up real-world applications as compared to current manual implementations.

To this end, we compare SMOKE to state-of-the-art logical and physical lineage capture and query approaches using microbenchmarks on single operator plans, as well as end-to-end evaluations over a subset of TPC-H queries. Using TPC-H, we further show that our workload-aware optimizations can provide further lineage query speedups on the "Overview first, zoom and filter, and details on demand" interaction paradigm and respond within interactive latencies of < 150ms [7, 38, 41]. Finally, we express two real-world applications (cross-filter [14] and data profiling [56]) in lineage terms and show that SMOKE *can match or outperform hand-optimized implementations of the same applications.*

Data. The microbenchmarks use a synthetic dataset of tables $\texttt{zipf}_{\vartheta,\text{n},\text{g}}$`(id,z,v)` containing zipfian distributions of varying skew. `z` is an integer that follows a zipfian distribution and `v` is a double that follows a uniform distribution in $[0, 100]$. ϑ controls the zipfian skew, n is the table size, and g specifies the number of distinct z values (i.e., groups). Tuple sizes are small to emphasize worst-case lineage overheads. End-to-end and workload-aware experiments use the TPC-H data generator and vary the scale factor. Our experiments on real-world applications use the Ontime [45,46] (123.5m tuples, 12GB) and Physician [49] (2.2m tuples, 0.6GB) datasets.

To ensure a fair comparison, we implement and optimize alternative, state-of-the-art techniques in our query engine. Our implementation reduces the capture overheads (by several orders of magnitude) as compared to their original implementations, and is detailed in our technical report [50].

First, we describe the compared lineage capture techniques (see also Table 1 for a brief description of the techniques):

SMOKE-based techniques. **SMOKE-I** and **SMOKE-D** instrument the plan using INJECT and DEFER instrumentation (Section 3). Unless otherwise noted, SMOKE-I and SMOKE-D don't use optimizations from Section 3. **BASELINE** evaluates base queries on SMOKE without capturing lineage.

Baseline logical techniques. State-of-the-art logical approaches (PERM [22], GPROM [44]) use query rewrites to annotate the base query output with lineage. However, they are built on production databases that incur overheads from transaction and buffer managers, lack of hash-table reuse, and lack of query compilation. These factors could confound results on a system-to-system comparison on the lineage capture problem and would not lead to meaningful results. For this reason, we implemented PERM's rewrite rules (and

GProm's optimizations, whenever applicable) in Smoke to generate physical plans that annotate output records with either rids (**Logic-Rid**) or full input tuples (**Logic-Tup**). As we noted in Section 2, the output annotated relations need to be indexed to support fast lineage lookups. To this end, **Logic-Idx** scans the annotated output relation to construct the same end-to-end lineage indexes as those created by Smoke. For completeness, we also note that our implementation of logical approaches in Smoke are two orders of magnitude faster than with Perm and GProm. (Details on how we optimized logical techniques in Smoke are in [50].)

Baseline physical techniques. To highlight the importance of tightly integrating lineage capture and operator logic, we use two baseline physical techniques. **Phys-Mem** instruments each operator to make virtual function calls to store input-output rid pairs in Smoke lineage indexes from Section 3, which highlights the overhead of making a virtual function call for each lineage edge. **Phys-Bdb** instead indexes lineage data in BerkeleyDB to showcase the drawbacks of using a separate storage subsystem [59].

Moreover, we compare lineage querying techniques based on data models and indexes induced during lineage capture:

Lineage consuming queries. Smoke-I, Smoke-D, Logic-Idx, and Phys-Mem all capture the same lineage indexes from Section 3.1, thus their lineage consuming query performance will be identical. We call this group **Smoke-L**. We compare with a baseline lazy approach, **Lazy**, which uses standard rules [15, 28] to rewrite lineage consuming queries into relational queries that scan the input relations. We also compare with the data model that Logic-Rid and Logic-Tup produce and the indexes that Phys-Bdb generate. Finally, we consider Lazy and Smoke without optimizations as baselines to our workload-aware optimizations.

Settings for the real-world applications are provided inline.

Measures. For lineage capture, we report the absolute base query latency and relative overhead compared to not capturing lineage. For lineage and lineage consuming queries, we report absolute latency and speedup over baselines. All numbers are averaged over 15 runs, after 3 warm-up runs.

Platforms. We ran experiments on a MacBook Pro (macOS Sierra 10.12.3, 8GiB 1600MHz DDR3, 2.9GHz Intel Core i7), and a server-class machine (Ubuntu 14.04, 64GiB 2133MHz DDR4, 3.1GHz Intel Xeon E5-1607 v4). Both architectures have caches sizes 32KiB L1d, 32KiB L1i, and 256KiB L2—the MacBook has 4MiB L3 and the server-class has 10MiB L3. Our overall findings are consistent across the two architectures. Since lineage capture is write-intensive, we report results on the lower memory bandwidth setting (MacBook). Only for crossfilter, we report server-class results because the Ontime dataset exceeds the memory of MacBook.

6. EXPERIMENTAL RESULTS

In this section, we first compare lineage capture techniques on microbenchmarks (Section 6.1) and TPC-H queries (Section 6.2). Then, we compare techniques on lineage query evaluation (Section 6.3) and showcase the impact of our workload-aware optimizations (Section 6.4). We conclude with experiments on real-world applications (Section 6.5).

6.1 Single Operator Lineage Capture

We first evaluate lineage capture with a set of single operator microbenchmarks for group-by (Section 6.1.1), pk-fk joins (Section 6.1.2), and m:n joins (Section 6.1.3).

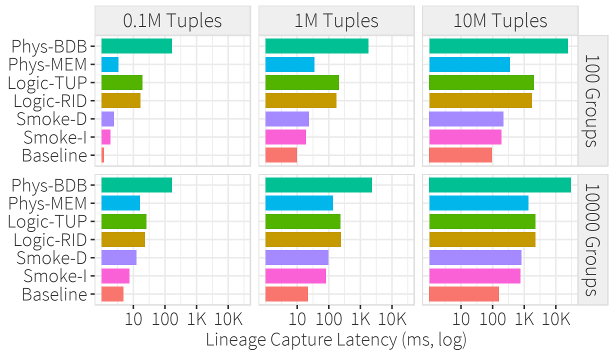

Figure 4: Comparison of lineage capture costs for the group-by aggregation operator for different relation cardinalities (columns) and number of distinct groups (rows). Smoke-I and Smoke-D slow down the non-instrumented Baseline the least as compared to alternative logical and physical capture methods.

6.1.1 *Group-by Aggregation*

We use the following base query, which groups by z drawn from a zipfian distribution so that cardinalities are skewed:

```
Q$_{\doteq}$= SELECT    z, COUNT(*), SUM(v), SUM(v*v),
                        SUM(sqrt(v)), MIN(v), MAX(v)
      FROM      zipf$_{\vartheta=1,n,g}$
      GROUP BY  z -- #groups follow a zipfian
```

$Q_{\doteq}$ computes multiple statistics following visualization systems that group multiple statistics in a single query [55].

Figure 4 reports the lineage capture latency (base query latency + capture overhead) for each technique while varying the input size (columns) and number of groups (rows).

Smoke. Smoke-I incurs the lowest overhead among techniques (0.7× on average). Smoke-D is slightly slower (1.2× on average) due to the cost of its join $\bowtie_{\gamma}$ for lineage capture.

Comparison with logical techniques. Logic-Rid and Logic-Tup use Perm's aggregation rewrite rule, which computes $Q_{\doteq} \bowtie_z$ `zipf` to derive the denormalized lineage graph as a single relation. The cost of computing and writing the denormalized lineage graph is costly, slows the base query by multiple orders of magnitude, and is one of the main reasons why Smoke outperforms alternative logical techniques. Furthermore, since `zipf` is narrow, Logic-Tup performs similarly to Logic-Rid. However, we expect Logic-Tup to perform worse for wider input relations. Logic-Idx has extra indexing costs over Logic-Rid and is not plotted.

Comparison with physical techniques. The primary overhead for Phys-Mem is the cost of a virtual function call for each written lineage edge. The cost of building index data structures is comparable to Smoke's write costs, however Smoke can reuse the hash table built by γ'_{ht} and incur lower costs for building the backward lineage rid index. Phys-Bdb incurs by far the highest overhead (up to 250× slowdown), due to the overhead of communicating with BerkeleyDB. The same trends hold for the other operators and we have not found physical approaches to be competitive. *As such, we do not report physical approaches in the rest of the experiments.*

Varying dataset size, skew, and groups. In general, the lineage capture techniques all incur a constant per input tuple overhead, and differ on the constant value. This is why increasing the input relation size increases costs linearly for all techniques. Increasing the number of groups increases the costs of building and scanning the group-by hash table as well as the output cardinality, and affects all techniques including the baseline. We find that the overhead is independent of the zipfian skew because it does not change the number of

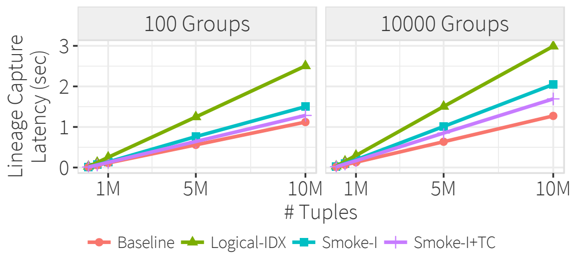

Figure 5: SMOKE-I reduces the instrumented pk-fk join latency from 1.4× (LOGIC-IDX) to 0.41×. Knowing the join cardinalities further reduces the overhead to 0.23× (SMOKE-I-TC). SMOKE-D is equivalent to SMOKE-I for pk-fk joins.

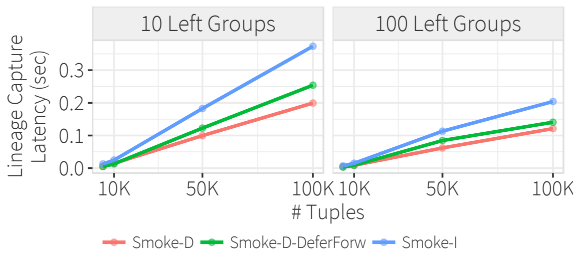

Figure 6: M:N join latency when all indexes are populated with SMOKE-I, only forward indexes for the left table are deferred (SMOKE-D-DEFERFORW), and when both lineage indexes are deferred for the left table (SMOKE-D).

lineage edges that need to be written. The skew does affect querying lineage, however, as we will see in Section 6.3.

Complexity of group-by keys and aggregate functions. We find that the techniques differ in their sensitivity to the size of the group-by keys and the number of aggregation functions in the project clause of the query. SMOKE-I simply generates rid indexes and rid arrays, and is not affected by these characteristics of the base query. In contrast, SMOKE-D and both logical approaches are sensitive to the size of the group-by keys, since they are used to join the output and input relations. Finally, the logical approaches are also affected by the number of aggregation functions because they affect the cost of the final projection. In short, we believe our setup is favorable to alternatives and conclude that SMOKE still shows substantial lineage capture benefits.

Cardinality Statistics. SMOKE can also leverage group cardinalities (e.g., through histograms) to allocate correctly sized lineage indexes (Section 3). This further reduces the capture overhead by 52% on average and leads to overhead reduction from 0.7× to 0.3× for SMOKE-I (not plotted).

6.1.2 *Primary-Foreign Key (Pk-Fk) Joins*

We evaluate lineage capture on pk-fk joins with the query $Q_{\Join}$ = `SELECT * FROM gids,zipf WHERE gids.id=zipf.z`. `zipf.z` is a foreign key that references `gids.id` and is drawn from a zipfian distribution ($\vartheta = 1$) so that some keys contribute to more join outputs than others. We vary the number of join matches by varying the unique values for `gids.id`. In addition to BASELINE and SMOKE-I, we evaluate SMOKE-I-TC which assumes that we know the number of matches for each `gids.id` and highlights the costs of array resizing. Note that SMOKE-D is equivalent to SMOKE-I due to the pk-fk optimizations in Section 3.2.4. We compare against LOGIC-IDX because LOGIC-RID and LOGIC-TUP do not support forward queries without additional indexes.

Comparison with logical techniques. LOGIC-IDX incurs 1.4× capture overhead on average due to the costs of computing and materializing the denormalized lineage graph in the form of the annotated output relation, and scanning the annotated table to build backward and forward lineage indexes for both input relations. In contrast, SMOKE-I incurs on average 0.41× overhead; knowing join cardinalities reduces the overhead to 0.23× on average. Finally, note that SMOKE-I already knows the cardinalities for the backward indexes and the forward index of the right table for pkfk joins (Section 3.2.4). Thus, the lower overhead of SMOKE-I-TC is due to lower reallocation costs for the forward index of the left table—which is the most expensive index to build due to the 1-to-N relation between primary keys and join outputs.

6.1.3 *Many-to-Many (M:N) Joins*

We evaluate lineage capture on M:N joins with the query $Q_{\Join}$=`SELECT * FROM zipf1,zipf2 WHERE zipf1.z=zipf2.z` that performs a join over the two `z` attributes drawn from zipfian distributions ($\vartheta = 1$). `zipf1.z` is within $[1, 10]$ or $[1, 100]$ while `zipf2.z`$\in [1, 100]$. This means that tuples with $z = 1$ have a disproportionately large number of matches compared to larger z values that have fewer matches. For this experiment, we also fix the size of the left table `zipf1` to 10^3 records and vary the right `zipf2` from 10^3 to 10^5.

Section 3.2.4 described the INJECT approach for M:N joins, which populates the lineage indexes within the probe phase ($\Join_{\text{probe}}$), and the DEFER approach, which computes cardinality statistics during the probe phase to correctly allocate and populate the lineage indexes for the left table after the probe phase to avoid array resizing costs. Finally, to show the benefits of DEFER, we also evaluate SMOKE-D-DEFERFORW which still defers the forward index construction for the left table but populates the backward index within the probe phase. To simplify the presentation, we only report SMOKE-based techniques since our comparisons with alternatives yields findings consistent with the ones presented so far.

Comparison of SMOKE techniques. M:N joins over the skewed inputs of our setup are similar to cross-products and yield very large output relations. As a result, the join output materialization dominates the base query execution and renders the lineage capture overheads non-informative. For this reason, here we present results without accounting for the materialization of the output. In this way, the M:N execution is ≈ 0ms and Figure 6 primarily reports lineage capture overhead for the three techniques that we compare. The overheads for SMOKE-I and SMOKE-D-DEFERFORW is predominantly due to resizing. SMOKE-D avoids resizing and reduces the capture overhead the most (up to 2.65×). Finally, increasing the number of groups for `zipf1.z` reduces the costs of all techniques because the output cardinality is smaller but the relative capture overheads are the same.

Other Operators. Our technical report [50] describes additional results and operators. The main additional finding is that it is preferable to overestimate selection cardinality estimates to avoid array resizings for the selection operator.

6.2 Multi-Operator Lineage Capture

To evaluate lineage capture on multi-operator plans, we used four queries from TPC-H (i.e., Q1, Q3, Q10, and Q12). Their physical query plans contain group by aggregation as the root operator, selections that vary in predicate complexity and selectivity, and up to three pk-fk joins. (Our hash-based execution precludes sort operations.) Figure 7 summarizes

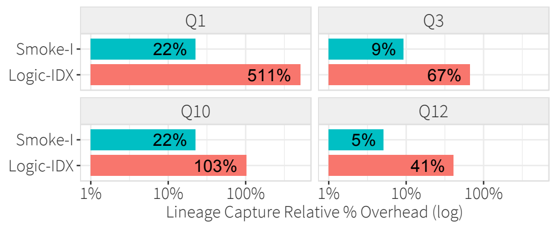

Figure 7: Relative overhead of SMOKE and logical lineage capture techniques for TPC-H queries Q1, Q3, Q10, and Q12. (SF=1)

the overhead of the best performing SMOKE (i.e., SMOKE-I) and logical (i.e., LOGIC-IDX) techniques for the four queries.

Overall Results. SMOKE-I reduces the capture overhead as compared to LOGIC-IDX by up to 22×. In addition, SMOKE-I incurs at most 22% overhead across the four queries. To ensure that the reported overhead results are meaningful, we made sure that the query engine of SMOKE has reasonable performance. Despite its row-oriented execution, SMOKE is comparable to MonetDB (single-threaded, data cached in OS buffers): Q1 runs in 176ms while the slowest query Q12 runs in 306ms.[2] SMOKE-D (not shown) is slower than SMOKE-I due to the cost of $\bowtie_{\Upsilon}$ for lineage capture on the aggregation operator. However, it is still faster than the logical approaches. (We refer interested readers to our technical report [50] for a more detailed discussion.) Finally, although Q1 is simple (e.g., it has no joins), its results are arguably the most informative because its selections have the highest selectivity, which most stresses overheads as we discuss next.

Impact of selections in lineage capture. We found that the selectivity of the query predicate has a large impact on the overhead of logical approaches. Q1 introduces a setting where the predicate has a high selectivity. Thus, the input to the final aggregation operator has a high cardinality. This leads output groups to depend on a large set of input records which, in turn, results in a large amount of duplication in the denormalized representation of the lineage graph. The other queries have low predicate selectivity which leads to lower (albeit significant) data redundancy. Overall, SMOKE is not sensitive to this effect because the lineage indexes represent the normalized lineage graph to avoid data duplication.

Lineage Capture Takeaways (Sections 6.1 and 6.2). *SMOKE-based lineage capture techniques outperform both logical and physical alternatives by up to two orders of magnitude. Logical approaches that adhere to the relational model are affected by the denormalized lineage graph representation, extra indexing steps, and expensive joins. Physical approaches are affected by virtual function calls and write-inefficient lineage indexes. Array resizing contributes to a large portion of SMOKE overheads. However, accurate or overestimated statistics can further reduce resizing costs (up to 60%).*

6.3 Lineage Query Performance

We now evaluate the performance of different lineage query techniques. Recall that lineage queries are a special case of lineage consuming queries. We evaluate the query: `SELECT * FROM` L_b(o∈ $Q_{\pm}$`(zipf), zipf`), where $Q_{\pm}$(zipf) is the query used in the group-by microbenchmark (Section 6.1.1) and o denotes an output group. For this experiment, $Q_{\pm}$`(zipf)` contains 5000 groups while `zipf` contains 10M records and we vary its skew ϑ. Varying ϑ highlights the query performance

[2]The purpose is *not* to compare SMOKE with MonetDB, but to ensure that the reported overheads are over a reasonable baseline.

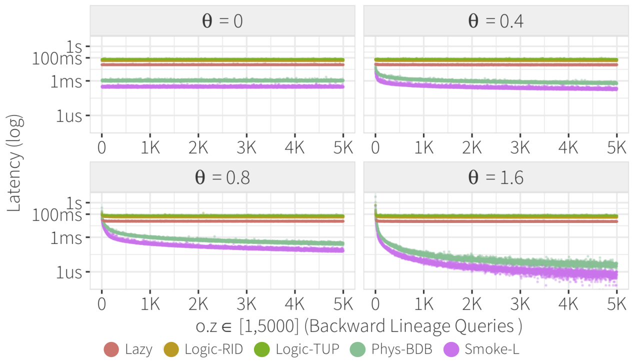

Figure 8: Lineage query latency for varying data skew (ϑ). LAZY has a fixed cost to scan the input relation and evaluates a selection on the group-by key `o.z=?`. LOGIC-RID and LOGIC-TUP perform the same selection but on annotated output relations. SMOKE-L is mainly around 1ms and outperforms LAZY, LOGIC-RID, and LOGIC-TUP by up to five orders of magnitude for low selectivity lineage queries. The crossover points at high selectivities are due to the costs of SMOKE-L index scans. SMOKE-L is a lower bound for PHYS-BDB that incurs extra costs for reading from inefficient lineage indexes and communicating with external lineage subsystems.

with respect to the cardinality of the backward lineage query. Figure 8 reports the lineage query latency for all 5000 o assignments and different ϑ values (i.e., $\vartheta \in \{0, 0.4, 0.8, 1.6\}$).

Recall that when we capture lineage with SMOKE-I; SMOKE-D; LOGIC-IDX; or PHYS-MEM, we evaluate lineage queries with SMOKE-L. SMOKE-L evaluates the lineage queries of our setup above using secondary index scans (i.e., it uses the contributing input rids of an output o from the backward index of $Q_{\pm}$ to perform lookups into `zipf`). Next, we compare SMOKE-L with lazy, logical, and physical alternatives.

Comparison with LAZY. In contrast to SMOKE-L, LAZY performs a table scan of the input relation and evaluates an equality predicate on the integer group key. This is arguably the cheapest predicate to evaluate and constitutes a strong comparison baseline. We find that SMOKE-L outperforms LAZY up to five orders of magnitude, particularly when the cardinality of the output group is small. We expect the performance differences to grow when the base query uses more complex group-by keys, which increases the predicate evaluation cost, or when the input relation is wide, which increases scan costs [9, 32, 35]. Finally, there is a cross over point when the input relation is highly skewed ($\vartheta \in \{0.8, 1.6\}$) and the backward lineage of some groups have high cardinality. This increases the secondary index scan cost of SMOKE-L in comparison to the serial scans of LAZY.

Comparison with logical techniques. We also report the cost of scanning the annotated relations generated by LOGIC-RID and LOGIC-TUP (highest two lines). Scanning these relations to answer lineage queries is worse than LAZY because the annotated relation is wider than the input relation, yet they have the same cardinality. This is the main reason why we introduced extra indexing steps for the annotated output relations of logical approaches with LOGIC-IDX. (Recall that LOGIC-IDX is represented here by SMOKE-L.)

Comparison with physical techniques. PHYS-MEM is included as part of SMOKE-L, so we report PHYS-BDB. Using an external lineage subsystem to perform a lineage query, we need to perform function calls to the external system to fetch the input rids for an output group o. As long as we have the input rids, we can perform a secondary index scan

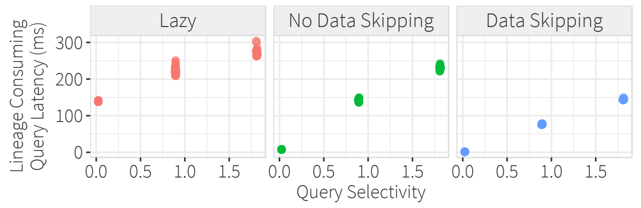

Figure 9: Lineage consuming query latency for different instrumentation approaches as the lineage consuming query's selectivity varies. Lazy requires table scans, No Data Skipping performs more efficient secondary index scans, and Data Skipping is $\leq$ 150ms because it only scans the relevant partition of the lineage index.

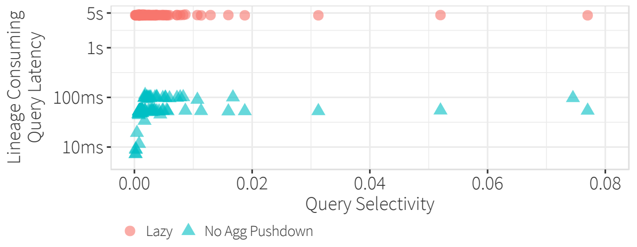

Figure 10: SMOKE-I reduces the lineage consuming query latency by 72.9× on average as compared to LAZY. With aggregation push-down, the latency is ≈ 0ms and we do not plot it.

to evaluate the lineage query similarly to SMOKE-L. In our experiments, we compared both fetching all input rids in a single function call as well as with consecutive function calls in a cursor-like fashion. The cursor-like approach outperformed the bulk approach since it avoids allocation costs for input rids. SMOKE-L provides a lower bound for PHYS-BDB: both perform the same secondary index scan but PHYS-BDB pays the cost of function calls to the external subsystem and it depends on indexes with worse read performance.

Lineage Query Takeaways: *SMOKE outperforms logical and lazy lineage query evaluation strategies by multiple orders of magnitude, especially for low-selectivity lineage queries. We believe SMOKE is a lower bound for physical approaches by avoiding functions calls and using read-efficient indexes.*

6.4 Workload-Aware Optimizations

We explore the effectiveness of the data skipping and group-by push-down optimizations by incrementally building up an example motivated by the "Overview first, zoom and filter, details on demand" [54] interaction paradigm. We focus only on zoom and filter because the base query generates the initial overview, while details on demand is the simple backward lineage query evaluated in Section 6.3. We report selection push-down and pruning in our technical report [50].

We use TPC-H Q1 as the initial "Overview" base query (SF=2), and we render its output groups as a bar chart. There are four bars generated from 48%, 24%, 24%, and 0.06% of the `Lineitem` relation. Subsequent interactions (e.g., zoom in by drilling down and filter by adding predicates) will be expressed as lineage consuming queries that incrementally modify their preceding lineage consuming queries.

No optimization. Before considering optimizations, we first assess the effectiveness of lineage indexes on the evaluation of lineage consuming queries as compared to the lazy approach. Suppose users are interested in drilling into a particular bar to see its statistics grouped further by the month and year of the shipping date. This is expressed as a lineage consuming query $\mathrm{Q1_a}$ that changes Q1 in two ways: (1) replaces the input relation with the backward lineage of the bar (i.e., $\mathrm{L_b(o_a \in Q1(}$`Lineitem`$\mathrm{),}$`Lineitem`$\mathrm{))}$ and (2) adds `Month,Year` of the shipping date to the `GROUP BY` clause.

We evaluate $\mathrm{Q1_a}$ for every value of $\mathrm{o_a}$ (not plotted). LAZY runs $\mathrm{Q1_a}$ as a table scan followed by filtering on Q1's group-by keys, grouping on year and month, and computing the same aggregates as Q1. SMOKE-I executes the same steps but evaluates $\mathrm{Q1_a}$ with secondary index scans as opposed to table scans. SMOKE-I performs best when the group cardinality is low (0.06% selectivity) and outperforms LAZY by 6.2×. For higher cardinality groups, SMOKE-I incurs the overheads of secondary index scans, as we also noted in Section 6.3. However, the performance of the two methods is similar because processing the high lineage cardinality (to compute the group-by aggregations in this case) dominates the execution of $\mathrm{Q1_a}$. A principal approach to avoid such high processing costs is using our workload-aware optimizations.

Data skipping. Suppose we know that the users want to filter the result of Q1 (e.g., based on interactive filter widgets). Then we can push this logic into lineage capture using the data skipping optimization. We evaluate $\mathrm{Q1_b}$, which extends $\mathrm{Q1_a}$ with two parameterized predicates: `l_shipmode = :p1 AND l_shipinstruct = :p2`. Q1 is the base query for $\mathrm{Q1_b}$. To exercise push-down overheads, both are text attributes and thus more expensive to evaluate than numeric attributes. The lineage capture overhead was 0.22× for SMOKE-I and 1.65× with the data skipping optimization due to the additional cost of partitioning the rid arrays on the text attributes, but still lower than logical approaches (Figure 7).

Figure 9 plots the lineage consuming query latency for the selectivities of every possible combination of the predicate parameters. The LAZY baseline executes the lineage consuming query as a filter-groupby query over a table scan of `Lineitem`. Although lineage indexes substantially reduce query latency (No Data Skipping in Figure 9)—particularly for low predicate selectivities—it is bottlenecked by the secondary scan costs of backward lineage for high cardinality groups. In contrast, Data Skipping reduces even high selectivity queries by at least 2× compared to LAZY, and is consistently below the interactive 150ms threshold [38]. This is because rid arrays are partitioned by `l_shipmode, l_shipinstruct` and the lineage consuming query is evaluated using indexed scans with only the rids needed to correctly answer the query.

Group-by aggregation push-down. After users filter and identify interesting statistics from the filter interactions in $\mathrm{Q1_b}$, they may want to drill down further. If we know this upfront, SMOKE can pre-compute aggregates for new dimensions with the group-by aggregation push-down optimization. To evaluate this optimization, we compare LAZY against SMOKE-I (with and without the optimization) on $\mathrm{Q1_c}$. $\mathrm{Q1_c}$ changes $\mathrm{Q1_b}$ by adding `l_tax` to the `GROUP BY` clause and setting the input relation to $\mathrm{L_b(o_c \in Q1_b(...),}$`Lineitem`$\mathrm{)}$. For this experiment, we consider $\mathrm{Q1_b}$ as the base query of $\mathrm{Q1_c}$.

Figure 10 compares the lineage query latency under LAZY (red dots) against SMOKE-I without the optimization (blue triangles). The push-down optimization is not plotted because it takes ≈ 0ms (i.e., just fetches the materialized aggregates). For completeness we vary the parameters of the backward lineage statement $\mathrm{L_b()}$ for $\mathrm{Q1_c}$ ($\mathrm{L_b(o_c \in Q1_a, ...)}$) as well as for the base query $\mathrm{Q1_a}$ ($\mathrm{L_b(o_a \in Q1, ...)}$) of $\mathrm{Q1_b}$ and report the lineage consuming query's latency for all combinations. Overall, LAZY takes $>$ 4 seconds per $\mathrm{Q1_c}$ instance while

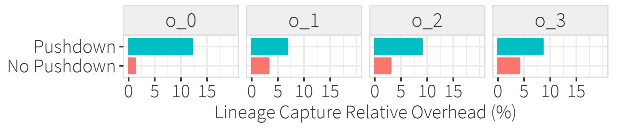

Figure 11: The average relative instrumentation overhead increases from 2.9% without to 9.15% with aggregation push-down.

SMOKE-I takes from 7ms to 100ms without the optimization and $\approx$ 0ms with the optimization for all $Q1_c$ instances.

Pre-computing aggregation statistics is not free, however. Figure 11 plots the lineage capture overhead for both SMOKE variants over to the non-instrumented lazy approach. We report the result for all 4 parameters to the base query $Q1_a$'s backward lineage statement ($L_b(o_a \in Q1, ...)$). The overhead of SMOKE-I is low compared to the overall cost of partitioning the rid arrays on `l_tax` and computing aggregates.

Push-down Takeaways: *Our experiments highlight that lineage indexes are sufficient whenever the lineage cardinality is low for the complexity of future lineage consuming queries. For higher lineage cardinalities, our workload-aware optimizations provide a principled way to push-down computation into lineage capture and optimize future lineage consuming queries. They also highlight trade-offs that future optimizers would need to consider (see also open questions in Section 4.2).*

6.5 SMOKE-Enabled Applications

We now present evidence that SMOKE can optimize real-world applications to the extent that it can perform on par with or even improve on hand-tuned, application-specific implementations. More specifically, we show how SMOKE optimizes cross-filter visualizations (Section 6.5.1) and data profiling (Section 6.5.2) primitives. We highlight the main results here and defer details to our technical report [50].

6.5.1 Crossfilter

Crossfilter is an important interaction technique to help explore correlated statistics across multiple visualization views [14]. In the common setup, multiple group-by queries along different attributes of a dataset are each rendered as, say, bar charts. (Each bar chart corresponds to a visualization view.) When a user highlights a bar (or set of bars) in one view, the other views update to show the group-by results over only the subset that contributed to the highlighted bar (or bars). This is naturally expressed as backward lineage from the highlighted bar, followed by refreshing the other views by executing the group-by queries on the lineage subset.

Since the views are fundamentally aggregation queries, recent research proposals construct variations of data cubes to accelerate the crossfilter interactions [37, 39, 47]. However, it can take minutes or hours to construct these data cubes. Such offline time is not available if a user has loaded a new dataset (e.g., into Tableau) and wants to explore using cross-filter as soon as possible. This has recently been referred to as the cold-start problem for interactive visualizations [4].

Setup. Following previous studies [37, 39, 47], we used the Ontime dataset and four group-by `COUNT` aggregations on <lat, lon> (65,536 bins), <date> (7,762 bins), <departure delay> (8 bins), and <carrier> (29 bins); only 8,100 bins have non-zero counts because <lat, lon> is sparse. Each group-by query corresponds to one output view. This setup favors cube construction because it involves only four views and coarse-grain binning on spatiotemporal dimensions (which decreases the size of cubes and increases group cardinalities). We report the individual (Figure 12) and cumulative (Figure 13) latency to highlight each and every bar, respectively.

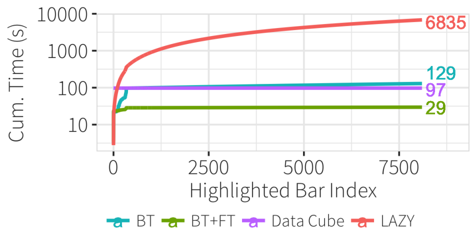

Figure 12: Cumulative latency of different crossfiltering techniques. BT+FT outperforms all approaches with the total time to perform the initial group-by aggregates, track lineage, and evaluate all interactions being thirty seconds.

Techniques. We compare the following: **LAZY** uses lazy lineage capture and re-executes the group-by queries on the lineage subset. **BT** uses SMOKE to capture backward lineage indexes but re-runs the group-by queries (which requires re-building group-by hash tables). **BT+FT** also captures forward lineage indices that map input records to the output bars that they contribute to, which can be used to incrementally update the visualization bars without re-building group-by hash tables. Finally, we compare with **DATA CUBE** construction. We first ran IMMENS [39], NANOCUBES [37], and HASHEDCUBES [47] to construct the data cubes. However, IMMENS and NANOCUBES did not finish within 30 minutes, while HASHEDCUBES required 4 minutes. For this reason, we implemented a custom partial cube construction based on our group-by aggregation push-down optimization that took 1.6 minutes to construct. This construction resembles the low dimensional cube decomposition described by IMMENS but using the sparse encoding recommended by NANOCUBES.

Main Results. We make four main observations. First, we observe that BT outperforms LAZY by leveraging the backward index to avoid table scans; BT+FT outperforms BT because the forward index lets SMOKE directly update the associated visualization bars without the need to re-build group-by hash tables; and, although the **DATA CUBE** response time is near-instantaneous, the cube construction cost is considerable and BT+FT is able to complete the benchmark before the cube is constructed (Figure 12). Second, BT+FT performs best ($<$ 10ms) when group-by queries output many groups (e.g., lat/lon and date) because then each group's backward lineage is substantially small. This suggests that lineage can complement cases when data cubes are expensive (e.g., when a cube dimension contains many bins) by computing the results online. Third, Figure 13 shows that BT+FT responds within $<$ 150ms (dotted line) for all but five bars, whose lineage depends on a large subset of the input tuples ($>$10% selectivity; $>$13M tuples). Fourth, the capture overhead for BT+FT and BT on the initial group-by queries are relatively low ($<$ 2$\times$ using SMOKE-I). We expect optimizations that use parallelization, sampling, and deferred lineage capture to reduce crossfilter latencies even further.

6.5.2 Data Profiling Applications

Data profiling studies the statistics and quality of datasets, including constraint checking; data type extraction; or key identification. Recent work, such as UGUIDE [56], proposes human-in-the-loop approaches towards mining and verifying functional dependencies (FD), by presenting users with examples of potential constraint violations to double-check.

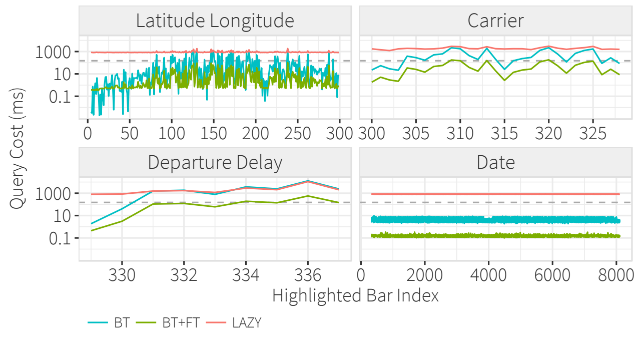

Figure 13: Latency for each crossfilter interaction. Dashed lines correspond to 150ms interaction layer. BT+FT performs under the 150ms interaction layer for all 8,100 but 5 interactions, with interactions on the spatiotemporal dimensions to be <10ms. Data Cube has instantaneous response time and we do not plot it.

This experiment compares UGUIDE with SMOKE on a lineage-oriented specification of an interactive data profiling problem.
Setup. We evaluate the following task: given a functional dependency (FD) A → B over a table T and an FD evaluation algorithm that outputs the distinct values a ∈ A that violate the FD, our goal is to construct a bipartite graph that connects the violations a with the tuples $\{t \in T \mid t.A = a\}$. Collectively, for a set of given FDs, this construction leads to a two-level bipartite graph connecting FDs and violations to tuples responsible for the violations. We compare SMOKE-based approaches with UGUIDE's implementation[3]. Based on correspondence with the authors, it turns out that UGUIDE internally creates data structures akin to the lineage indexes that SMOKE captures. This makes sense because it mirrors a lineage-based technique for the problem, as we show next.
Techniques. FD violations for A → B can be identified by transforming the FD into one or more SQL queries. We consider two rewrite approaches. The simple approach (**CD**) runs the query Q_{cd}=`SELECT A FROM T GROUP BY A HAVING COUNT(DISTINCT B) > 1`; backward and forward lineage indexes for Q_{cd} correspond to the desired bipartite graph above. Now, UGUIDE implements an optimization which, *although not modeled as lineage, effectively simulates lineage indexes.* We thus describe the second approach (**UG**) in lineage terms. We first evaluate the query $Q_{ug,attr}$=`SELECT DISTINCT attr FROM T` for `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 there are more than one distinct B values in the forward traced output, then the FD is violated. The lineage indexes also correspond to the desired bipartite graph. The **UG** approach is typically faster than the **CD** one for FD mining because **UG** builds lineage indexes once per attribute and reuses them across FD checks. Our experiments report the costs for individual FD checks and the bipartite graph construction. However, we note that the relative findings are expected to grow wider for multi-FD checks.

Next, we compare SMOKE that implements both approaches (SMOKE-CD,SMOKE-UG) with UGUIDE that implements the **UG** approach in METANOME (METANOME-UG).
Main Results. Figure 14 evaluates the techniques using four functional dependencies over the Physician dataset used

[3]UGUIDE proposed novel algorithms for mining and verifying functional dependencies. Although latency was not the focus of UGUIDE, its METANOME [48]-based implementation is optimized for latency. So, we believe it is a reasonable comparison baseline.

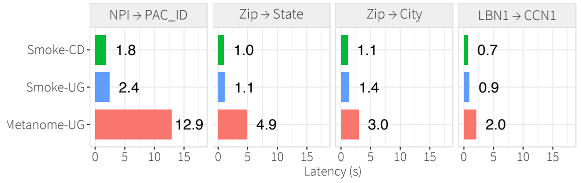

Figure 14: Latency of different approaches for FD violation evaluation and bipartite graph construction. SMOKE-CD is the minimal overall. METANOME-UG is affected by virtual function calls for lineage capture, the overheads of JVM, and its data model.

in the Holoclean [51] paper. Overall, SMOKE-UG outperforms METANOME-UG by $2-6\times$ while the simpler SMOKE-CD approach outperforms both approaches. Both SMOKE capture overheads are consistent with our microbenchmarks ($< 1.2\times$ overhead). There are several reasons why SMOKE-UG outperforms METANOME-UG. METANOME-UG incurs virtual function call costs when constructing its version of lineage indexes ($> 2\times$ overhead on $Q_{ug,attr}$ that we implemented in UGUIDE), as well as general JVM overhead even after a warm-up phase to enable JIT optimizations. Furthermore, METANOME-UG models all attribute types as strings, which slows uniqueness checks for integer data types such as `NPI`. For fairness, the other three FDs are over string attributes (zip is a string).

Application Takeaways: *Lineage can express many real-world tasks, such as those in visualization and data profiling, that are currently hand-implemented in ad-hoc ways. We have shown evidence that lineage capture can be fast enough to free developers from implementing lineage-tracing logic without sacrificing, and in many cases, improving performance.*

7. CONCLUSIONS AND FUTURE WORK

SMOKE shows that it is possible to quickly capture lineage and quickly answer lineage queries. SMOKE reduces the overhead of fine-grained lineage capture by avoiding shortcomings of logical and physical approaches in a principled manner, and is competitive or outperforms hand-optimized visualization and data profiling applications. SMOKE also contributes to the space of physical database design by being the first system to use lineage for physical design decisions. Our capture techniques and workload-aware optimizations may be used for online, adaptive, and offline physical designs. Finally, we believe the design principles **P1-P4** are broadly applicable beyond the design of our engine.

There are many areas for future work to explore: 1) leverage modern features such as vectorized and compressed execution, columnar formats, and UDFs [52], 2) develop cost-based techniques to instrument plans in an application-aware manner (e.g., DEFER is best-suited for speculation in-between interactions), 3) model database optimization policies (e.g., statistics computation, cube construction, key indexes) as lineage queries, and 4) extend support to data cleaning, visualization, machine learning, and what-if [3, 17] applications.

Acknowledgments. Our reviewers provided thoughtful and thorough reviews. Saravanan Thirumuruganathan shared with us the UGuide implementation and provided invaluable support. We thank Molham Aref, Boris Glavic, Goetze Graefe, Joe Hellerstein, Christoph Koch, Sanjay Krishnan, Neokli Polyzoti, Thibault Sellam, Val Tannen, and Yifan Wu for many useful discussions. This work is supported by NSF grants #1564049 and #1527765.

8. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[3] S. Assadi, S. Khanna, Y. Li, and V. Tannen. Algorithms for provisioning queries and analytics. *CoRR*, abs/1512.06143, 2015.

[4] L. Battle, R. Chang, J. Heer, and M. Stonebraker. Position statement: The case for a visualization performance benchmark. In *DSIA*, 2017.

[5] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[6] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[7] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *CHI*, pages 181–186, 1991.

[8] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, pages 445–456, 2014.

[9] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, pages 361–372, 2003.

[10] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou. Data provenance at internet scale: architecture, experiences, and the road ahead. In *CIDR*, 2017.

[11] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[12] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: A post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[13] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *PVLDB*, 9(12):1137–1148, 2016.

[14] Crossfilter. `http://square.github.io/crossfilter/`, 2015.

[15] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *TODS*, 25(2):179–227, 2000.

[16] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5):577–588, 2017.

[17] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[18] K. Dursun, C. Binnig, U. Cetintemel, and T. Kraska. Revisiting reuse in main memory database systems. In *SIGMOD*, pages 1275–1289, 2017.

[19] EU GDPR. `https://www.eugdpr.org/`.

[20] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends®in Databases*, 8(1-2):1–130, 2017.

[21] Facebook folly. `http://bit.ly/fbfolly`, 2017.

[22] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[24] T. J. Green and V. Tannen. The semiring franework for database provenance. In *PODS*, pages 93–99, 2017.

[25] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. *PVLDB*, 8(12):2004–2007, 2015.

[26] N. Hanusse, S. Maabout, and R. Tofan. Revisiting the partial data cube materialization. In *ADBIS*, pages 70–83, 2011.

[27] J. Heer, M. Agrawala, and W. Willett. Generalized selection via interactive query relaxation. In *CHI*, pages 959–968, 2008.

[28] R. Ikeda. *Provenance In Data-Oriented Workflows*. PhD thesis, Stanford University, 2012.

[29] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.

[30] R. Ikeda and J. Widom. Panda: A system for provenance and data. *Data Engineering Bulletin*, 33(3):42–49, 2010.

[31] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[32] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[33] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *PVLDB*, 7(10):797–808, 2014.

[34] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.

[35] M. S. Kester, M. Athanassoulis, and S. Idreos. Access path selection in main-memory optimized data systems: Should i scan or should i probe? In *SIGMOD*, pages 715–730, 2017.

[36] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, pages 1717–1722, 2017.

[37] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *EuroVis*, 19(12):2456–2465, 2013.

[38] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *TVCG*, 20(12):2122–2131, 2014.

[39] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013.

[40] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *SoCC*, pages 17:1–17:15, 2013.

[41] R. Miller. Response time in man-computer conversational transactions. In *AFIPS*, pages 267–277, 1968.

[42] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[43] T. Neumann and V. Leis. Compiling database queries into machine code. *Data Engineering Bulletin*, 37(1):3–11, 2014.

[44] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. Provenance-aware query optimization. In *ICDE*, pages 473–484, 2017.

[45] Airline On-Time Performance. `http://stat-computing.org/dataexpo/2009/the-data.html`.

[46] Airline On-Time Statistics and Delay Causes. `https://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp`.

[47] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *TVCG*, 23(1):671–680, 2017.

[48] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. *PVLDB*, 8(12):1860–1863, 2015.

[49] Physician Compare National. `https://data.medicare.gov/data/physician-compare`.

[50] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *ArXiv e-prints*, abs/1801.07237, 2018.

[51] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[52] A. Rheinländer, U. Leser, and G. Graefe. Optimization of complex dataflows with user-defined functions. *CSUR*, 50(3):38:1–38:39, 2017.

[53] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *PVLDB*, 9(4):348–359, 2015.

[54] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Symposium on Visual Languages*, pages 336–343, 1996.

[55] P. Terlecki, F. Xu, M. Shaw, V. Kim, and R. Wesley. On improving user response times in tableau. In *SIGMOD*, pages 1695–1706, 2015.

[56] S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J.-A. Quiane-Ruiz, and N. Tang. Uguide: User-guided discovery of fd-detectable errors. In *SIGMOD*, pages 1385–1397, 2017.

[57] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[58] E. Wu, S. Madden, and M. Stonebraker. A demonstration of dbwipes: clean as you query. *PVLDB*, 5(12):1894–1897, 2012.

[59] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, pages 865–876, 2013.

[60] E. Wu, F. Psallidas, Z. Miao, H. Zhang, L. Rettig, Y. Wu, and T. Sellam. Combining design and performance in a data visualization management system. In *CIDR*, 2017.

[61] Z. Zhang, E. R. Sparks, and M. J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *HPDC*, pages 143–153, 2017.