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 has served as an integral ingredient for both traditional (e.g., debugging, auditing, data integration, and security) and emergent (e.g., interactive visualizations, iterative analytics, explanations, and cleaning) applications. The core, long-standing problem that lineage systems need to address—and the main focus of this paper—is to capture the relationships between input and output data items across a workflow with the goal to streamline queries over lineage. Unfortunately, current lineage systems either incur high lineage capture overheads, or lineage query processing costs, or both. As a result, applications, that in principal can express their logic declaratively in lineage terms, resort to hand-tuned implementations. To this end, we introduce Smoke, an in-memory database engine that neither lineage capture overhead nor lineage query processing needs to be compromised. To do so, Smoke introduces tight integration of the lineage capture logic into physical database operators; lineage storage in efficient, write-optimized lineage representations; and optimizations when future lineage queries are known up-front. Our experiments on microbenchmarks and realistic workloads show that Smoke reduces the lineage capture overhead and streamlines lineage queries by multiple orders of magnitude compared to state-of-the-art alternatives. Our experiments on real-world applications highlight that Smoke can meet the latency requirements of interactive visualizations (e.g., <150ms) and outperform hand-written implementations of data profiling primitives.

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; forward queries return the subset of output records that depend on a given subset of input records.

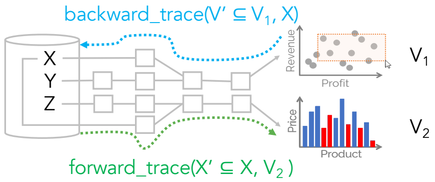


Figure 1: Two workflows generate visualization views V_1 and V_2. Then, a linked brushing interaction highlights in red marks in V_2 that share the same input records with selected marks of V_1. This interaction can be expressed as a backward query from selected circles in V_1 followed by a forward query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

Virtually, any application that requires an understanding over the input-output derivation process can be expressed in lineage terms. As such, data lineage has been an integral ingredient for applications such as debugging [18,46,50,58,89], data integration [22], auditing [27], security [16,50], explaining query results [23,78,87,88], data cleaning [12,37], iterative analytics [19], and interactive visualizations [90] that highlight the importance of lineage-enabled systems.

Lineage-enabled systems answer lineage queries by automatically capturing record-level relationships throughout a workflow. A naive 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 [2,9,32,44] or explicit pointers [58,89]). Each design trades off between the time and storage overhead to capture lineage and lineage query performance. For instance, a query execution engine may augment each operator to materialize a hash index that looks up input records for a given output record, thus speeding up backward lineage query execution. However, the overhead of constructing the index can dwarf the operator execution cost by 100× or more [89]—particularly if the operator is heavily optimized for latency or throughput.

As data processing engines become faster, an important question—and the main focus of this paper—is whether it is possible to achieve the best of both worlds: negligible lineage capture overhead *as well as* fast lineage query execution.

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:

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 (e.g., circles) in one view, and marks derived from the same records are highlighted in the other views. Although this functionality is typically implemented manually, it can be logically expressed as a backward lineage query from selected points in V_1 to input records followed by a forward query to highlight the corresponding bars in V_2.*

[*] A version of this paper has been accepted to VLDB 2018 (camera ready pending). This document is its associated technical report and has not been peer-reviewed in its entirety.

To avoid the shortcomings of current lineage systems and tackle the competing requirements of lineage applications, we employ a careful combination of four design principles:

P1. Tight integration. In high throughput query processing systems, per-tuple overheads incurred within a tight loop—even a single virtual function call to write lineage metadata to a separate lineage subsystem [46, 58, 89]—can slow down operator execution by more than an order of magnitude. In response, we introduce a physical algebra that tightly integrates lineage capture into query execution, and we design simple, write-efficient data structures for lineage capture to avoid the overhead of crossing system boundaries.

P2. Apriori knowledge. Lineage applications such as debugging need to capture lineage to answer ad-hoc lineage queries that can trace back to any base or intermediate table. In applications such as interactive visualizations or data profiling we typically know the possible set of lineage queries up front. Having apriori knowledge enables us to avoid materializing lineage that does not contribute to these lineage queries.

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 based on templated analysis (e.g., Tableau or Power BI), physical design logic based on these templates can be pushed into the lineage capture phase. Such physical design logic may include materialization of aggregate statistics or prune/re-partition lineage indexes to speed up future lineage consuming queries.

P4. Reuse. Finally, we have found that significant lineage capture costs arise from generating and storing unnecessary amounts of lineage data such as expensive annotations and denormalized forms of lineage. Following the concept of reusing data structures [26], we identify cases where data structures constructed during normal operator execution can be augmented and reused, but this time with the goal set to capture lineage with low overhead.

This paper presents Smoke, a lineage-enabled system that embodies the above four principles to support lineage capture and querying with low latency. More specifically, Smoke is an in-memory query compilation database engine that tightly integrates the lineage capture logic within query execution and uses simple, write-efficient lineage indexes for low-overhead lineage capture (**P1**). In addition, Smoke enables workload-aware optimizations that prune captured lineage and push the logic of lineage consuming queries down into the lineage capture phase (**P2**,**P3**). Finally, Smoke identifies data structures, which are constructed during normal operator execution (i.e., hash tables), and (re)uses them, whenever possible, for low-overhead lineage capture (**P4**).

In the rest of the paper, we start by discussing necessary background and related work (Section 2). Then, we present our techniques and contributions as follows:

- We introduce a physical algebra that tightly integrates the lineage capture logic within the processing of single and multi-operator plans. Each logical operator has a dual form to both execute its logic and generate lineage. In turn, physical operators are combined to implement this extended logic. Furthermore, we design write-efficient data structures to materialize lineage with low overhead.(Section 3)
- We design a suite of simple optimizations based on the availability of future lineage consuming queries to 1) prune lineage that will not be used and 2) materialize aggregates and prune or re-partition our lineage data structures to answer lineage consuming queries faster. (Section 4)
- We conduct experiments to 1) compare Smoke with state-of-the-art lineage systems and 2) show how it can enable real-world applications (i.e., interactive visualizations and data profiling). The former experiments show that Smoke reduces the lineage capture overhead and streamlines lineage queries by multiple orders of magnitude compared to state-of-the-art lineage systems. The latter suggest that Smoke lets applications express their logic using declarative lineage constructs *and also* speeds up these applications—to the extent that Smoke is on par with or outperforms hand-written implementations. (Sections 5 and 6)

We conclude with open questions for future work (Section 7).

2. BACKGROUND

In this section, we provide background on the fine-grained lineage capture problem, the approach of Smoke for this problem, and applications of focus in this paper.

2.1 Fine-Grained Lineage Capture

Our lineage semantics adhere to the transformational provenance semantics of [19, 32, 43] over relational queries.

Base queries. Formally, let the *base query* $Q_{⏚}(D) = O$ 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 $\mathbb{Q}_{⏚} = \{Q_{⏚1}, \cdots, Q_{⏚m}\}$. For instance, $\mathbb{Q}_{⏚}$ in Figure 1 consists of two queries that generate two output relations rendered as 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 is 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(•)`. 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]

Example 2. *Let $Q_{⏚1}(\{X, Y\}) = V_1$ and $Q_{⏚2}(\{X, Z\}) = V_2$ 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, similarly to the ones in [90].*

We can model interactive visualization applications as base queries (e.g., $Q_{⏚1}$, $Q_{⏚2}$ above) that load the initial visualization, followed by lineage consuming queries W that express user interactions [90]. Therefore, optimizing the visualization responsiveness corresponds to quickly executing the base queries followed by streamlining lineage consuming queries.

[1] Smoke's query model includes multi-backward and multi-forward queries as well as refresh and forward propagation [43]. We limit the discussion to L_b, L_f, and C in that they form the basis to express general query constructs.

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 overhead at the cost of potentially slower lineage query execution costs [17, 22, 43]. In contrast, we might *Eagerly* materialize data structures during base query execution to speed up future lineage queries [17,43]. We refer to this as lineage capture, and we seek to minimize capture overhead on the base query to speed up lineage queries.

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 help speed up lineage consuming queries.

We will present techniques that can efficiently capture lineage indexes 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. In general, lineage capture techniques fall into two categories: logical and physical.

Logical lineage capture. This class of approaches stay within the relational model by rewriting the base query into $Q'_{\underline{\top}}(\{R_1, \cdots, R_n\}) = O'$, so that its output is annotated with additional attributes of input tuples. Some systems [2, 18] 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 [18, 32] output a single denormalized representation that extends O' with attributes of the input relations. Recent work has shown that the latter rewrite rules (Perm [32]) and optimizations leveraging the database optimizer (GProm [66]) 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—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 approach directly instruments physical operators write lineage edges to a lineage subsystem through an API; the subsystem stores and indexes the edges and answers lineage queries [44, 45, 46, 58, 89]. This can support black-box operators and decouples lineage capture from its physical representation. However, we find that virtual function calls alone (ignoring the cross-process overheads) can slow data-intensive operators by up to 2×. Further, lineage capture cannot easily leverage and be co-optimized with 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 can exploit knowledge of lineage consuming queries to (a) prune or partition lineage indexes or (b) materialize views *during the base query execution* that will benefit the lineage consuming queries.

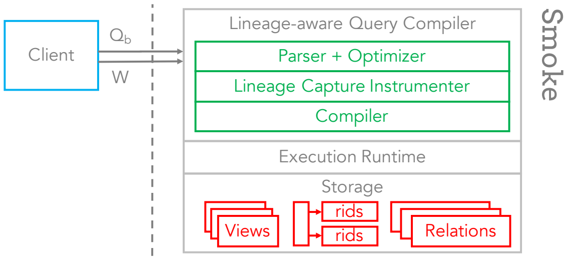


Figure 2: Smoke is a query compilation engine that instruments physical query plans to capture lineage efficiently: Query execution generates lineage indexes that map input and output record ids (rids) as well as materialized views.

The primary focus of this work is to explore mechanisms to instrument physical operator plans with lineage capture logic. To do so, we have implemented Smoke as a query compilation execution engine using the produce-consumer model [65] (Figure 2). It takes as input the base query $Q_{\underline{\top}}$ and an optional workload of lineage consuming queries W; parses and optimizes $Q_{\underline{\top}}$ to generate a physical query plan; instruments the plan to directly generate indexes to speed up backward and forward lineage queries; and compiles the instrumented plan into machine code that, when executed, generates $Q_{\underline{\top}}(D)$ as well as lineage indexes. 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. Execution models that use advanced features such as compression or vectorization are interesting future work.

2.3 Lineage Applications

Many applications logically rely on lineage (and generally provenance), including but not limited to: debugging [18, 46, 50, 58, 89], diagnostics [83], data integration [22], security [16, 50], auditing [27] (the recent EU GDP regulation [27] mandates tracking lineage), data cleaning [12, 37], explaining query results [23, 78, 87, 88], debugging machine learning pipelines [54, 91], and interactive visualizations [90].

Unfortunately, there is a disconnect between modeling applications in terms of lineage, and the performance of existing lineage capture mechanisms—the overhead is enough that applications resort to manual implementations instead. For this reason, we center the paper around interactive visualizations: it is a domain that can directly translate to lineage [40, 90], yet is dominated by hand-written implementations. Furthermore, it imposes strict latency requirements on lineage capture (to show the initial visualization) and lineage consuming queries (to respond to user interactions). Finally, our experiments seek to argue, using visualization and data profiling applications, that lineage is not only an elegant logical description of many use cases, but can be on a par with or even *improve* on performance compared to hand-tuned implementations.

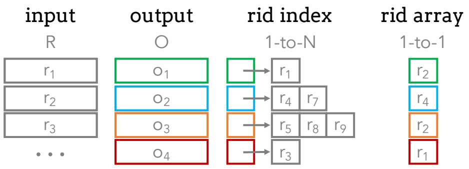


Figure 3: Lineage index representations: rid index for 1-to-N operators (e.g., GROUPBY); rid array for 1-to-1 operators (e.g., SELECT).

3. FAST LINEAGE CAPTURE

This section describes lineage capture without knowledge of the future workload. We present (a) lineage index representations to map output-to-input or input-to-output record ids (*rids*) that are read- and write-efficient, and (b) a physical algebra that tightly integrates the lineage capture logic with the base query execution. To do so, we will describe how to instrument individual as well as multiple operators to capture lineage with low overhead.

3.1 Lineage Index Representations

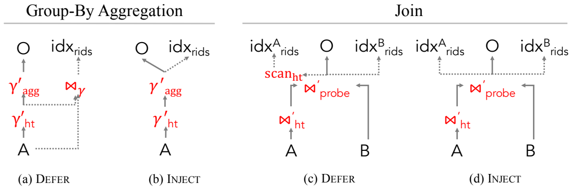
Smoke uses two main lineage index representations. Figure 3 illustrates the input and output relations R and O, respectively, and the two rid-based representations for 1-to-N and 1-to-1 operators. We index rids because the lineage indexes are cheap to write, and lookups—which simply index into the relation's array—are fast. In contrast, indexing full tuples incurs high write costs, while indexing primary keys is not beneficial without building primary key indexes apriori or when the primary keys are wide. Furthermore, in-memory columnar engines [1, 28] already create rid lists as part of query processing that resemble our lineage indexes, and enable reuse opportunities to reduce lineage capture costs.

Rid Index. 1-to-N relationships are represented as inverted indexes. Consider the backward lineage of GROUPBY. The index's i^{th} entry corresponds to the i^{th} output group, and points to an rid array containing rids of the input records that belong to the group. The Rid Index is used for 1-to-N forward lineage relationships as well, such as the JOIN operator. Following high performance libraries [29], 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, and statistics that allow Smoke to pre-allocate appropriate sized arrays can reduce lineage capture costs by up to 60%. To avoid offline statistics computation, we show how useful statistics can be collected during query processing.

Rid Array. 1-to-1 relationships between output and input are represented as a single array. Each entry is an input record rid rather than a pointer to an rid array.

3.2 Single Operator Instrumentation

We now introduce instrumentation techniques to generate lineage indexes when executing single and multi-operator plans. Our designs are based on two paradigms: Defer defers portions of the lineage capture until after operator execution while Inject incurs the full cost during execution. Defer is preferable when the base query execution overhead *must* be minimized, or when it is possible to collect cardinality statistics during base query execution to allocate appropriately sized lineage indexes and avoid resizing costs. In contrast, Inject typically incurs lower overall overhead, but the client needs to wait longer to retrieve the results of the base query.



(a) Defer (b) Inject (c) Defer (d) Inject

Figure 4: Inject and Defer plans for group-by aggregation and join. Dotted arrows are only necessary for lineage capture.

We now describe how both paradigms illustrate the application of the tight integration and reuse principles from the Introduction for core relational operators. Our focus is on the mechanisms and Section 7 discusses future work on automatically choosing between these paradigms. Code snippets of the compiled code and details for additional operators (including $\cup$, $\cap$, $-$, $/$, $\times$, $\bowtie_\theta$) are covered in our technical report [75]. Section 3.3 extends our support to multi-operator plans.

3.2.1 Projection

Projection under bag semantics does not need lineage capture because the input and output order and cardinalities are identical—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 below.

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 [64]. Both forward and backward lineage use rid arrays; the forward rid array can be pre-allocated 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 ctr_i^{th} element of the forward rid array to ctr_o, and append ctr_i to the backward rid array. Available selectivity estimates can be used to pre-allocate the backward rid array and avoid reallocation costs during the append operation. We don't implement Defer because it is strictly inferior to Inject.

3.2.3 Group-By Aggregation

Query compilers decompose `GROUPBY` into two physical operators: γ_{ht} builds the hash table that maps group-by values to the group's intermediate aggregation state; γ_{agg} scans the hash table, finalizes aggregation results for each group, and emits output records. Figure 4 shows the plans for both instrumentation paradigms; the lineage indexes consist of a forward rid array and a backward rid index.

Defer: Consider the Defer plan in Figure 4.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 are used to 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 4.b. γ'_{ht} this time augments each group's intermediate state with an rid array *i_rids* that contains the rids of the group's input records (i.e., backward lineage). γ'_{agg} tracks the current output record id *oid* in order to set the pointer in the backward index to the bucket's rid list and the values in the forward rid list. 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 in γ'_{ht}. We find that knowing group cardinalities can decrease the lineage capture overhead up to 60%.

3.2.4 Join

Smoke instruments hash joins in a similar way as hash aggregation. A hash join is split into two physical operators: $\Join_{ht}$ builds the hash table on the left relation A, and $\Join_{probe}$ uses each record of the right relation B to probe the hash table. We now introduce Inject and Defer techniques for lineage capture that can be used for general M:N joins, and further optimizations for primary key-foreign key (pk-fk) joins. Smoke generates backward rid arrays and forward rid indexes (an input record can generate multiple join results).

Inject: Consider the Inject plan for joins in Figure 4.d. The build phase $\Join'_{ht}$ 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 $\Join'_{probe}$ 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 $\Join'_{probe}$ phase and we cannot pre-allocate our lineage indexes. As a result, although the backward rid array is relatively cheap to resize, the forward rid indexes can potentially trigger multiple reallocations if an input record has many matches and penalize the performance.

Defer: Our main observation is that exact cardinalities needed to pre-allocate the forward rid indexes are known *after* the probe phase and can be used by Defer. Note that deferring the whole construction after the probe phase is similar to the logical approaches, and incurs the cost of re-running the join, or annotating, indexing and projecting the join output. Defer instead augments Inject by partially deferring index construction for the left input relation A (see Figure 4.c).

The build phase adds a second rid list *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 $\Join'_{\text{probe}}$ phase, the forward and backward indexes for the left relation A can then be pre-allocated and populated in a final scan of the hash table (scan_{ht} in Figure 4.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; since the join cardinality is the same as the foreign-key table cardinality, backward indexes are pre-allocated. Finally, join selectivity estimates can help pre-allocate the forward rid indexes.

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 use the indexes to trace backward or forward through the plan. This approach is correct and can be used to 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 needed.

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 $op_p(op_c(R)) = O$ 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 $op_c(R)$; op_c's lineage indexes can be garbage collected when not needed further.

To reduce lineage index materialization points, recall that database engines pipeline operators to reduce intermediate results by merging multiple operators into a single pipeline [64]. 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 above.

Implementation Details Our engine supports naive 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. We focus on pk-fk joins due to their prevalence in benchmarks and real-world applications, and because the Inject and Defer instrumentation for pk-fk joins are identical (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. Details are in our technical report [75].

4. WORKLOAD-AWARE OPTIMIZATIONS

Lineage applications such as interactive visualizations will often support a pre-defined set of interactions (e.g., filter, pan, tooltip, cross-filter [20]) 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 that is not queried, and generate lineage representations that directly speed up queries in W. To simplify the discussion, we will center each optimization around different classes of lineage consuming queries over the base query $Q_{\underline{\perp}} = \sigma_{o\_orderdate > \text{'2017-08-01'}}(orders \Join lineitem)$.

4.1 Instrumentation Pruning

Instrumentation pruning disables lineage capture if the lineage indexes will not be used by W. We present two types of pruning that do not generate lineage for specific input relations, and for backward/forward lineage.

Pruning input relations. Suppose the visualization only supports a tooltip interaction that shows detailed lineitem information when the user hovers over a visualization mark. This is expressed as a backward lineage query to `lineitem`. In this case, we can avoid capturing lineage for the `orders` table. In general, Smoke does not capture lineage for any relation not referenced in W.

Pruning lineage direction. 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 is 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 the users—instead they will *reduce* the result cardinality with further filter, transform, and/or aggregation operations. This is also the case for lineage consuming queries, and presents opportunities to push these reduction operations into the lineage capture logic. We present three simple push-down optimizations for fixed filter predicates, templated predicates, and aggregation operations, and then discuss the relationship between push-down optimizations and common provenance semantics.

Selection push-down. Visualizations often update metrics that summarize data that the user selects. For instance, the following query retrieves Christmas shipment order information for parts of the visualization that the user interacts with: $C = \sigma_{shipdate=`xmas'}(L_B(O' \subseteq Q_{\underline{\perp}}(D), \text{orders}))$. This optimization pushes the predicate `shipdate='xmas'` into the instrumented base query; before Smoke populates the backward lineage indexes, it checks whether the input tuple satisfies the predicate. If the predicate is on a GROUPBY key, Smoke can also avoid any lineage capture overhead for all other groups. This technique reduces lineage space requirements, and typically reduces capture overhead. However, if the predicate is expensive to evaluate (e.g., slow UDF), it is possible to introduce more capture overhead.

Data skipping using lineage. Push down selections require a static predicate, however interactive visualizations also use parameterized predicates. For instance, the user may use a slider to dynamically specify the filtered shipping date: $C = \sigma_{shipdate=:p1}(L_B(O' \subseteq Q_{\underline{\perp}}(D), \text{orders}))$. This pattern is ubiquitous in interactive visualizations and applies to faceted search, cross-filtering, zooming, or panning. Smoke pushes the parameterized predicate into lineage capture by partitioning the rid arrays (standalone and part of rid indexes) by the predicate attribute. For instance, Smoke would partition the rid arrays in the backward index for `orders` by the `shipdate` attribute, so that C only reads the rid partition matching the parameter `:p1`. This technique is applicable to categorical attributes and continuous attributes that can be discretized. This is almost always possible because user-facing output is ultimately discretized at pixel granularity [48].

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 technique works if the main difference between the base and lineage consuming query is the addition of grouping attributes. In effect, lineage capture generates data cubes to answer the linage consuming aggregation query. In contrast to building data cubes offline, which requires separate scans of the database, this approach piggy-backs on top of the base query's existing table scans. As with prior work [34,38,57], this technique supports algebraic and distributive functions (e.g., `SUM`, `COUNT`, and `AVG`), and we evaluate this optimization extensively in synthetic (Section 6.4) and real-world settings (Section 6.5.1).

Relationships with Provenance Semantics. We observe that popular provenance semantics (e.g., which [22, 81] and why [10] provenance) can be expressed as lineage consuming queries and pushed down using the above optimizations. In other words, Smoke can operate as a system with alternative provenance semantics depending on the given lineage consuming query. For space reasons, we include a brief discussion in our technical report [75].

Applying Optimizations. Choosing the appropriate optimizations, manually or automatically, each poses challenges that we leave to future work. "What language extensions (e.g., `CREATE BACKWARD INDEX ON (SELECT ...)`) are needed to capture lineage manually?" and "What cost models are needed to model the trade-offs for each capture and optimization choice?" constitute interesting research questions.

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 $< 150ms$ [11, 56, 61]. Finally, we express two real-world applications (cross-filter [20] and data profiling [83]) 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 $\texttt{zipf}_{\theta,n,g}$`(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 [67, 68] (123.5m tuples, 12GB) and Physician [74] (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

Abbreviation	Description
Smoke	
Baseline Smoke-D Smoke-I	Smoke without lineage capture Smoke with defer lineage capture Smoke with inject lineage capture
Logical	
Logic-Rid Logic-Tup Logic-Idx	Rid-based annotation Tuple-based annotation Indexing input-output relations
Physical	
Phys-Mem Phys-Bdb	Virtual emit function calls and no reuse Lineage capture using BerkeleyDB

Table 1: Lineage capture techniques used in our evaluation.

orders of magnitude) as compared to their original implementations, and is detailed in our extended report [75].

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

Smoke 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 [32], GProm [66]) use query rewrites to annotate the base query output with lineage. However, they are built on production databases that incur unneeded capture overheads from e.g., transaction and buffer managers, lack of hash-table reuse, no query compilation. For this reason, we used Perm's rewrite rules (and whenever applicable, GProm's optimizations) to generate physical plans that annotate the output with either rids (**Logic-Rid**) or full input tuples (**Logic-Tup**), and implemented the plans in Smoke. Note that the output relation needs 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. Ultimately, our implementations are two orders of magnitude faster than Perm and GProm but still incur capture overheads higher than Smoke.

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 [89].

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 [22, 43] 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.

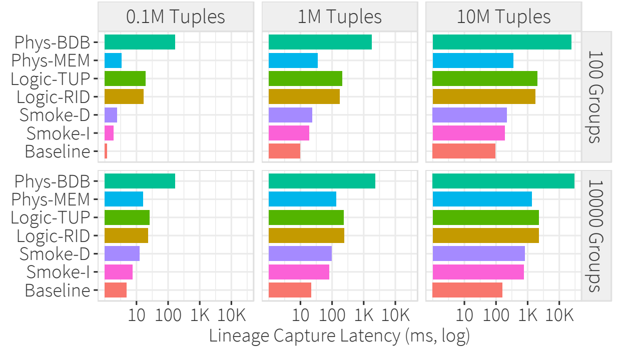


Figure 5: Comparison of lineage capture costs for the group-by aggregation operator for different relation cardinalities (columns) and number of distinct groups (rows). Smoke-I and Smoke-D slow down the non-instrumented Baseline the least as compared to alternative logical and physical capture methods.

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 for lineage capture are consistent across the two architectures. Since lineage capture is write-intensive, we report results using the lower memory bandwidth setting (MacBook). For the crossfilter application, we report the server-class results because the Ontime dataset doesn't fit in the MacBook memory.

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).

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 ($\theta = 1$). Visualizations often compute multiple statistics to avoid redundant scans when users ask for new statistics [82]:

```
Q⏚= SELECT     z, COUNT(*), SUM(v), SUM(v*v),
               SUM(sqrt(v)), MIN(v), MAX(v)
    FROM       zipf
    GROUP BY   z -- #groups follow a zipfian
```

Figure 5 reports the lineage capture latency (base query cost + overhead) for each instrumentation technique, and varies the input size (columns) and the 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 join to construct lineage indexes.

Comparison with `Logical` systems. Logic-Rid and Logic-Tup use Perm's aggregation rewrite rule, which computes $Q_{⏚} \bowtie_z$ `zipf` to derive the denormalized lineage graph as a single relation. The cost of computing and writing the denormalized lineage graph is costly and can slow the base query by multiple orders of magnitude. Since `zipf` is narrow, Logic-Tup performs similarly to Logic-Rid, however we expect the cost to increase for wider input relations. Logic-Idx has extra indexing costs over Logic-Rid and is not plotted.

Comparison with `Physical` systems. 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 lineage edges that need to be written; it does affect querying lineage 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 index 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 alternative approaches, and find that Smoke still shows substantial lineage capture benefits.

Cardinality Statistics. If Smoke knows the cardinality statistics for each group, then it can allocate correctly sized arrays in the lineage indexes (Section 3). This further reduces the lineage 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 use the following primary-foreign key join query: **`SELECT`** `* ` **`FROM`** `gids,zipf` **`WHERE`** `gids.id=zipf.z`. `zipf.z` is a foreign key that references `gids.id` and drawn from a zipfian distribution so that some keys are more popular than others. We vary the number of join matches (i.e., groups) by varying the number of 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 join attribute value—this is to highlight the costs of array resizing. Note that Smoke-D is equivalent to Smoke-I due to the pkfk optimization (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× relative 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 Smoke-I-TC's lower overhead can be attributed to lower reallocation costs to build the forward index for the left table.

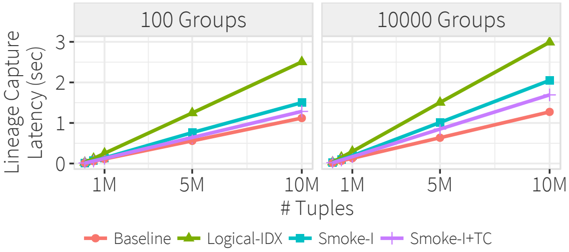


Figure 6: Smoke-I reduces the instrumented pk-fk join latency from 1.4× (Logic-Idx) to 0.41×. Knowing the join cardinalities further reduces the overhead to 0.23× (Smoke-I-TC). Smoke-D is equivalent to Smoke-I for pk-fk joins.

6.1.3 Many-to-Many Joins

For M:N joins we use the following base query that performs a join over two `z` attributes with zipfian distributions: **`SELECT`** `*` **`FROM`** `zipf1,zipf2` **`WHERE`** `zipf1.z=zipf2.z`. The join is highly skewed to stress lineage capture. Both `z` integer attributes are drawn from a zipfian distribution, where `zipf1.z` is within [1,10] or [1,100], while `zipf2.z`∈ [1,100]. This means that tuples with $z = 1$ have a disproportionate number of matches, whereas larger z will have few matches. Furthermore, we fix the size of the left table to 1000 records and vary the right from 10^3 to 10^5.

Section 3.2.4 described the Inject approach, which populates the lineage indexes within the join's probe phase ($\bowtie_{probe}$), and the Defer approach, which computes cardinality statistics during the probe phase to correctly allocate and populate the lineage indexes for the left table (`a_fw`,`a_bw`) after the probe phase and avoid array resizing costs. Finally, to break down the benefits of Defer, we also evaluate Smoke-D-DeferForw which still defers `a_fw` but populates `a_bw` within the $\bowtie_{probe}$ phase. To simplify the presentation, we only report Smoke-based techniques since the relationship with alternatives is consistent with the previous results.

Comparison of Smoke techniques. In contrast to the other operators, the MN join over skewed inputs is similar to a cross-product, and generates > 1billion results. Materializing the result set renders the capture overheads non-informative so we do not materialize the output. In this case, the MN execution is nearly 0, so Figure 7 primarily reports instrumentation overhead for the three techniques. The overhead is predominantly due to rid array resizing, which is why deferring the forward and backward lineage index construction for the left table reduces overhead by 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 their relative overheads are the same.

Other Operators. Our technical report [75] 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.

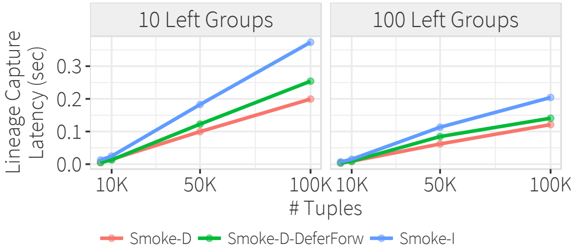


Figure 7: M:N join latency when all indexes are populated during the probe phase (Smoke-I), only forward indexes for the left table are deferred (Smoke-D-DeferForw), and when both lineage indexes are deferred (Smoke-D). These graphs highlight rid array resizing costs.

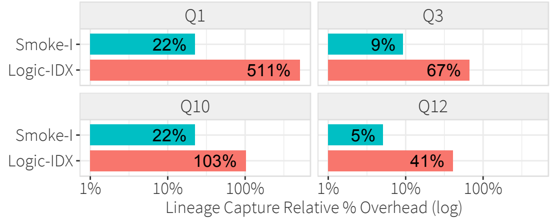


Figure 8: Relative overhead of Smoke and logical lineage capture techniques for TPC-H queries Q1, Q3, Q10, and Q12. Scale factor 1.

6.2 Multi-Operator Lineage Capture

We used four queries from TPC-H—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 8 summarizes the relative overhead of the best performing Smoke (i.e., Smoke-I) and logical (i.e., Logic-Idx) techniques for the four TPC-H queries. We use scale factor 1.

Overall Results. Smoke-I reduces the lineage 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 make the overhead results meaningful, we have tried to ensure that Smoke query engine has respectable performance—despite row-oriented execution it matches the performance of single-threaded MonetDB—non-instrumented 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 join cost between the input and output relations, however it is still faster than the logical approaches. Finally, although Q1 is simple (e.g., it has no joins), its results are arguably the most informative because the plan is simple and has the highest selectivity, which most stresses lineage capture.

Impact of selections in lineage capture. We found that the selectivity of the query predicate has a large impact on the overhead of the logical approaches. Q1 shows a setting where the predicate has a high selectivity, thus the input to the subsequent aggregation operator has a high cardinality, each output group depends on a larger set of input records, and ultimately leads a large amount of data duplication to create the denormalized lineage graphs. In contrast, the other queries have low predicate selectivity, thus the cardinality of the subsequent aggregation operator is small and leads to a substantially smaller lineage graph. Smoke is less sensitive to this effect because its lineage indexes represent a normalized lineage graph and avoids this duplication.

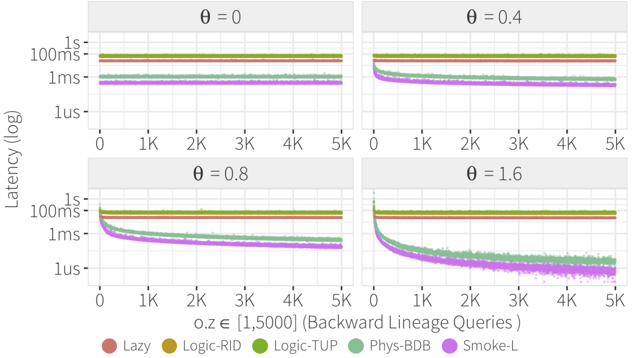


Figure 9: Lineage query latency for varying data skew (θ). Lazy has a fixed cost to scan the input relation and evaluates the simple selection predicate on the group-by key z=?. Logic-Rid and Logic-Tup performs the same selection but on annotated output relations. Smoke-L is mainly around 1*ms*, and outperforms Lazy, Logic-Rid, and Logic-Tup by up to five orders of magnitude for low selectivity lineage queries. The crossover point at high selectivities is due to the overhead of Smoke-L's index scan. Smoke-L is a lower bound for Phys-Bdb that incurs extra costs for reading lineage indexes.

Lineage Capture Takeaways (Sections 6.1 and 6.2): *Smoke-based techniques outperform both logical and physical approaches 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. The physical approaches are affected by virtual function calls and write-inefficient lineage indexes. We find that array resizing contributes to a large portion of* Smoke *overheads, and accurate or overestimated cardinality estimates can reduce resizing costs.*

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∈ $Q_{\underline{\perp}}$(zipf), zipf), where $Q_{\underline{\perp}}(zipf)$ is the query used in the group-by microbenchmark (Section 6.1.1). o is an output record (a group). zipf contains 5000 groups, $10M$ records, and we vary its skew θ. Varying θ highlights the query performance with respect to the cardinality of the backward lineage query. Figure 9 reports lineage query latency for all 5000 possible o assignments and different θ values.

Smoke-L evaluates the lineage query using a secondary index scan—it probes the backward index, and uses the input rids as array offsets into zipf. Recall that Smoke-L refers to any of Smoke-I, Smoke-D, Logic-Idx, or Phys-Mem (Section 5).

Smoke-L vs 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 keys that increase the predicate evaluation cost [14,47,52], or when the input relation is wide, which increases scan costs. There is a cross over point when the input relation is highly skewed ($\theta \in \{0.8, 1.6\}$) and the backward lineage of some groups have high cardinality. This increases Smoke-L's secondary index

[2] The purpose is *not* to compare Smoke with MonetDB, but to suggest that the reported overheads are with respect to reasonable baseline performance.

scan cost as compared to a linear table scan that can benefit from scan pre-fetching.

Smoke-L vs Logical Approaches. 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. Note that indexing the annotated relation is Logic-Idx, and represented by Smoke-L.

Smoke-L vs Physical Approaches. 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. As long as we have the input rids, we can perform a secondary index scan to evaluate the lineage query similarly to Smoke-L. In our experiments, we compare both fetching all input rids in a single function call, and with consecutive function calls to fetch the rids 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 lineage subsystem, and has worse lineage index read performance due to the B-Tree of BerkeleyDB.

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" [80] 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 [75].

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

No Optimization. We start off by evaluating the effectiveness of using lineage indexes (without optimizations) as compared to the lazy approach for lineage consuming queries (not plotted). Suppose the user will be interested in drilling into a particular bar to see the statistics broken down by month and year of the shipping date. This can be modeled as a lineage consuming query $Q1_a$ that extends Q1 in two ways: (1) replace the input relation with the backward lineage of the bar that the user is interested in (i.e., $L_b(o_a \in \mathrm{Q1}(\texttt{Lineitem}), \texttt{Lineitem})$) and (2) add `Month, Year` to the GROUP BY.

We evaluate $Q1_a$ for every value of o_a. Lazy runs $Q1_a$ 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 is best when the group cardinality is low (0.06% selectivity) and outperforms Lazy by 6.2×. Higher cardinality groups incur random seek overheads. The performance converges for high cardinality because the performance is dominated by the query processing costs (i.e., aggregation in this case). To address the overhead of high cardinality lineage queries, we next evaluate workload-aware optimizations.

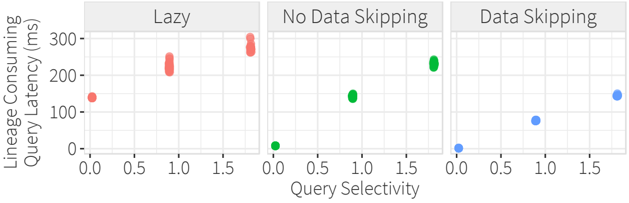


Figure 10: Lineage consuming query latency for different instrumentation approaches as the lineage consuming query's selectivity varies. Lazy requires full table scans, No Data Skipping performs more efficient secondary index scans, and Data Skipping is ≤ 150ms because it only scans the relevant partition of the lineage index.

Data skipping. Suppose we know that the user wants to filter the result of $Q1_a$ (say, based on interactive filter widgets), then we can push this logic into lineage capture using the data skipping optimization. We evaluate Q1$_b$, which extends Q1$_a$ with two parameterized predicates: `l_shipmode = :p1 AND l_shipinstruct = :p2`. Q1 is the base query for Q1$_b$. To exercise push-down overheads, both are text attributes and thus more expensive to evaluate than for 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 8).

Figure 10 plots the lineage consuming query latency vs the selectivity 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 10)—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 [56]. This is because rid arrays are partitioned by `l_shipmode, l_shipinstruct`, so that the lineage query only performs an indexed scan over the rids needed to answer the query.

Aggregation push down. After users filter and identify interesting statistics from the filter interactions in Q1$_b$, they may want to drill down further. If we know this upfront, Smoke may pre-compute aggregates for new dimensions during lineage capture. To this end, we define Q1$_c$ by adding `l_tax` to the group by clause in Q1$_b$, and setting the input relation to $L_b(o_c \in \mathrm{Q1}_b(...), \texttt{Lineitem})$. We compare Lazy (rewrites Q1$_c$ as a table scan-based query) against Smoke-I with and without the group-by optimization. In this setup, the previous lineage consuming query Q1$_b$ is the base query for Q1$_c$.

Figure 11 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 $L_b()$ for Q1$_c$ ($L_b(o_c \in \mathrm{Q1}_a, ...)$) as well as for the base query Q1$_a$ ($L_b(o_a \in \mathrm{Q1}, ...)$) of Q1$_b$ and report the lineage consuming query's latency for all combinations. Although Lazy takes > 4 seconds per Q1$_c$ instance, Smoke-I's index scan takes on average 100ms, and 10ms for low selectivity queries.

Pre-computing aggregation statistics is not free—Figure 12 plots the lineage capture latency for both Smoke variants

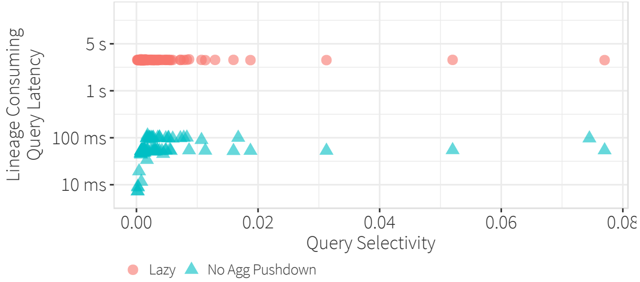


Figure 11: Smoke-I reduces the lineage consuming query latency by 72.9× on average as compared to Lazy. With aggregation push-down, the latency is $\approx$ 0ms and we do not plot it.

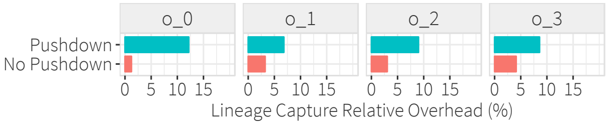


Figure 12: The average relative instrumentation overhead increases from 2.9% without to 9.15% with aggregation push-down.

as compared 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 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 consider (see also open research questions in Section 4.2).*

6.5 Smoke-Enabled Applications

We now present evidence that lineage can be used to optimize two real-world applications—cross-filter visualizations (Section 6.5.1) and data profiling (Section 6.5.2)—enough to perform on par with or better than hand-tuned, application-specific implementations. We highlight the main results here, and defer details to the technical report [75].

6.5.1 Crossfilter

Crossfilter is an important interaction technique to help explore correlated statistics across multiple visualization views [20]. In the common setup, multiple group-by queries along different attributes are each rendered as e.g., bar charts. When the 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(s). 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 have proposed variations of data cubes to accelerate the interactions [55, 57, 71], however it can take minutes or hours to construct the cubes. Such offline time is not available if the 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 [7].

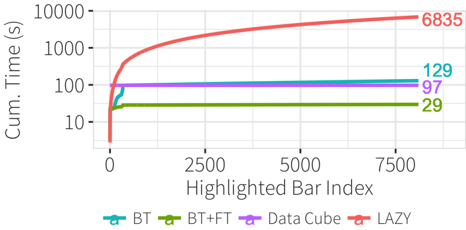


Figure 13: Cumulative latency of different crossfiltering techniques. BT+FT outperforms all approaches with the total time to perform the initial group-by aggregates, track lineage, and evaluate all interactions being thirty seconds.

Setup. Following previous studies [55, 57, 71], 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 number of cubes, and increases group cardinalities). We report the individual (Figure 13) and cumulative (Figure 14) latency to highlight each and every bar, respectively. The experiment was run on our server-class machine so that the dataset can fit in memory.

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 indices, but re-runs the group-by queries (which requires re-building group-by hashtables). **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-running the aggregation query. Finally, we compare with **Data Cube** construction. We first ran imMens [57], NanoCubes [55], and hashedcubes [71] 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, BT outperforms Lazy by leveraging the backward index to avoid table scans, and is consistent with our TPC-H benchmarks. Despite the overhead of forward index capture, BT+FT outperforms BT because the forward index lets Smoke directly update the associated visualization bars without the need to re-run aggregation queries (and re-build group-by hash tables). Although the **Data Cube** response time is near-instantaneous, the offline construction cost is considerable and BT+FT is able to complete the benchmark before the cube is constructed (Figure 13). Second, BT+FT performs best (< 10ms) when group-by queries output many groups (e.g., lat/lon, day) because they reduce group cardinality. This suggests that lineage can complement cases when data cubes are expensive (a cube dimension contains many bins) by computing the results online. Third, Figure 14 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 base visualization queries are relatively low (< 2× using

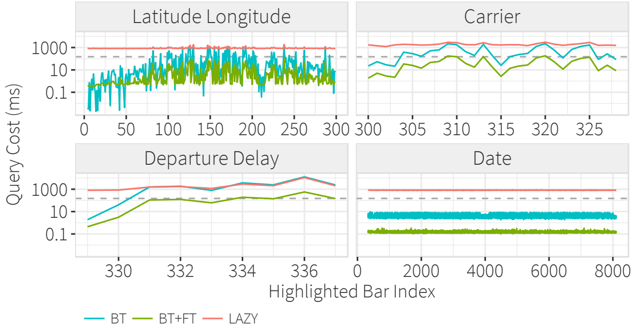


Figure 14: Latency for each 1D brushing crossfilter interaction. Dashed lines correspond to 150ms interaction layer. BT+FT performs under the 150ms interaction layer for all 8,100 but 5 interactions, with interactions on the spationatemporal dimensions to be <10ms. Data Cube has instantaneous response time and we do not plot it.

Smoke-I). We expect that optimizations that leverage parallelization, sampling, or deferred capture scheduled during user "think time" can further reduce the capture costs.

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 [83], proposes human-in-the-loop approaches towards mining and verifying functional dependencies, and present users with examples of potential constraint violations to double-check. This experiment compares UGuide with Smoke on a lineage-oriented specification of a data profiling problem.

Setup. We evaluate the following task: given a functional dependency (FD) $A \rightarrow B$ over a table T and an FD evaluation algorithm that outputs the distinct values $a \in A$ 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 description of the problem.

Techniques. FD violations for $A \rightarrow B$ 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 correspond to the bipartite graph above.

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 $Q_{ug,attr}$=`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 trace output, then the FD is violated. The lineage indexes also correspond to the desired bipartite graph.

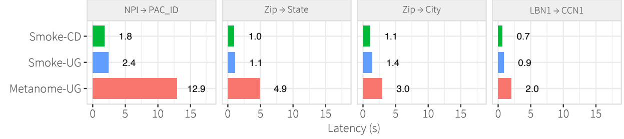


Figure 15: Latency of different approaches for FD violation evaluation and bipartite graph construction. Smoke-CD is the minimal overall. Metanome-UG is affected by virtual function calls for lineage capture, the overheads of JVM, and its data model.

The **UG** implementation is typically faster than **CD** for FD mining because **UG** explicitly builds lineage indexes once per attribute and reuses them across FD checks. Our experiments report the cost of individual FD checks and the cost to construct the bipartite graph, however the relative findings are expected to grow wider for multi-FD checks. We compare Smoke using both approaches (Smoke-CD, Smoke-UG) with UGuide that implements the **UG** approach (Metanome-UG).

Main Results. Figure 15 evaluates the techniques using four functional dependencies over the Physician dataset used in the Holoclean [76] paper. Overall, Smoke-UG outperforms Metanome-UG by 2 – 6× 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 optimization. Further, 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 illustrates that it possible to both capture lineage with low overhead and enable fast lineage query performance. Smoke reduces the overhead of fine-grained lineage capture by avoiding shortcomings of logical [9, 21, 31, 32, 35, 66, 86] and physical [43, 44, 45, 46, 58, 89] approaches in a principal manner, and is competitive or outperforms hand-optimized visualization and data profiling applications. Smoke also contributes to the space of physical database design [3, 4, 15, 25, 42, 51, 59, 62, 69, 73, 79, 84] by being the first engine to consider lineage as a type of information for physical design decisions. Our capture techniques and workload-aware optimization make Smoke well-suited for online; adaptive; and offline physical database design settings. Finally, we believe the design principles used in the development of Smoke (**P1-P4** in Introduction) are broadly applicable beyond our design.

There are many areas for future work to explore: 1) leverage modern features such as vectorized and compressed execution, columnar formats, and UDFs [77], 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 [6, 24] applications.

[3] UGuide proposed novel algorithms for mining and verifying functional dependencies, and implemented a fast version of the data-profiling task using Metanome [72]. Although latency was not their focus, the system was optimized for performance, so we believe it is a reasonable comparison baseline.

8. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 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*, 2006.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, 2014.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.

[6] S. Assadi, S. Khanna, Y. Li, and V. Tannen. Algorithms for provisioning queries and analytics. *arXiv preprint arXiv:1512.06143*, 2015.

[7] L. Battle, R. Chang, J. Heer, and M. Stonebraker. Position statement: The case for a visualization performance benchmark. In *DSIA*, 2015.

[8] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, 2016.

[9] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

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

[11] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *ACM*, 1991.

[12] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, 2014.

[13] R. Chang, M. Ghoniem, R. Kosara, W. Ribarsky, J. Yang, E. Suma, C. Ziemkiewicz, D. Kern, and A. Sudjianto. WireVis: Visualization of categorical, time-varying data from financial transactions. In *VAST*, 2007.

[14] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, 2003.

[15] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[16] 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.

[17] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. In *Foundations and Trends in Databases*, 2009.

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

[19] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *VLDB*, 2016.

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

[21] Y. Cui. *Lineage tracing in data warehouses*. PhD thesis, Stanford University, 2001.

[22] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 2000.

[23] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. In *VLDB*, 2017.

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

[25] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, 2005.

[26] K. Dursun, C. Binnig, U. Cetintemel, and T. Kraska. Revisiting reuse in main memory database systems. *arXiv preprint arXiv:1608.05678*, 2016.

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

[28] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends® in Databases*, 2017.

[29] Facebook folly. `https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md`, 2017.

[30] K. E. Gebaly and J. Lin. Afterburner: The case for in-browser analytics. *arXiv*, 2016.

[31] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, 2006.

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

[33] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[34] 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. In *Data Mining and Knowledge Discovery*, 1997.

[35] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[36] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, 2015.

[37] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. In *VLDB*, 2015.

[38] N. Hanusse, S. Maabout, and R. Tofan. Revisiting the partial data cube materialization. In *ADBIS*, 2011.

[39] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *SIGMOD*, 2008.

[40] J. Heer, M. Agrawala, and W. Willett. Generalized selection via interactive query relaxation. In *CHI*, 2008.

[41] J. M. Hellerstein, M. Stonebraker, J. Hamilton, et al. Architecture of a database system. *Foundations and Trends® in Databases*, 2007.

[42] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

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

[44] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *ICDE*, 2013.

[45] R. Ikeda and J. Widom. Panda: A system for provenance and data. In *IEEE Data Eng. Bull.*, 2010.

[46] 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. In *VLDB*, 2015.

[47] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. In *VLDB*, 2008.

[48] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. In *VLDB*, 2014.

[49] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *AVI*, 2012.

[50] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, 2010.

[51] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, 2005.

[52] 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*, 2017.

[53] Y. Klonatos, T. Rompf, C. Koch, and H. Chafi. Legobase: Building efficient query engines in a high-level language, 2014.

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

[55] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. In *EuroVis*, 2013.

[56] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. In *Vis*, 2014.

[57] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, 2013.

[58] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *SoCC*, 2013.

[59] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: adaptive partitioning for distributed joins. In *VLDB*, 2017.

[60] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*.

[61] R. Miller. Response time in man-computer conversational transactions. In *Fall joint computer conference*, 1968.

[62] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[63] F. Naumann. Data profiling revisited. *SIGMOD Rec.*, 2014.

[64] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[65] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE DEB*, 2014.

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

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

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

[69] Oracle. Oracle database 10g: The self-managing database. Technical report, Oracle, 2003.

[70] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

[71] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. In *TVCG*, 2017.

[72] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with metanome. In *VLDB*, 2015.

[73] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD*, 2015.

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

[75] F. Psallidas and E. Wu. Smoke: Fine-grained lineage capture at interactive speed. Technical report, Department of Computer Science, Columbia University, 2017.

[76] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. In *VLDB*, 2017.

[77] A. Rheinländer, U. Leser, and G. Graefe. Optimization of complex dataflows with user-defined functions. *ACM Comput. Surv.*, 2017.

[78] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. In *VLDB*, 2015.

[79] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden. Amoeba: a shape changing storage system for big data. In *VLDB*, 2016.

[80] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages*. IEEE, 1996.

[81] V. Tannen. The semiring franework for database provenance. In *PODS*, 2017.

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

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

[84] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, 2017.

[85] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *TVCG*, 2017.

[86] J. Widom. Trio: a system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[87] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *PVLDB*, 2013.

[88] E. Wu, S. Madden, and M. Stonebraker. A demonstration of dbwipes: clean as you query. In *VLDB*, 2012.

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

[90] 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.

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

APPENDIX

Our appendix material cover several details that we did not include in the main body as well as extensions of our techniques, discussion of open problems, additional experiments, and related work. More specifically, we first provide background on the query compilation model that Smoke employs (Appendix A) and how we tuned techniques (Appendix B). Furthermore, we provide a more detailed description of (a) the TPC-H Q1 variants, that we used in our experiments with workload-aware optimizations, (Appendix C) and (b) the crossfiltering techniques (Appendix D). In addition, we show how our workload optimizations can be used to encode different provenance semantics (Appendix E) and how Smoke performs lineage capture for bag and set union; intersection; and difference as well as lineage capture with nested-loop evaluation to support $\theta-$joins and cross products (Appendix F). We conclude with additional experiments (Appendix G) and related work (Appendix H).

A. QUERY COMPILATION

One of the main design principles that we realized in Smoke is the tight integration of the lineage capture logic with the query execution logic. In this section, we give a brief background on the query compilation and push-based execution model that Smoke leverages to realize this principle. (However, note that our techniques are not bound to this execution model. Based on this execution model we better describe tuning techniques of alternative logical and physical approaches for lineage capture in the next section. Readers familiar with query compilation concepts can skip to the next section.)

Query compilation combines query optimization and execution with low-level compiler-based optimizations (e.g., LLVM, LSM, GCC). It replaces query interpretation [41] with a *compilation phase* that transpiles queries into intermediate representations (IR) such as C, C++, or LLVM IR, that are further optimized by a standard compiler, and an *execution phase* than runs the query as a binary executable. Each operator in the physical plan emits its intermediate representation that implements its logic, and the engine emits glue code to combine operator inputs and outputs. Many modern database systems [5, 30, 36] have taken this approach, and continues to be an active research area [53] with positive results.

Query compilation systems typically implement operators using the *producer-consumer* code generation model [64] to derive a push-based execution model that is in contrast to the pull-based iterator [33], batch [70], or full-column [60] execution models of traditional query interpreters. Each operator exposes two functions: `produce` triggers child operators to produce tuples or data structures with the appropriate schema, and `consume` emits execution logic to process its inputs and hand the result to the parent consume methods. Borrowing the example from Neumann [64], the following generates pseudo code for the σ and scan operators:

```
σ.produce        σ.input.produce
σ.consume(a,s)   print 'if '+σ.condition;
                 σ.parent.consume(a, σ)
scan.produce     print 'for t in relation'
                 scan.parent.consume(attrs, scan)
```

The compilation phase will call the root operator's `produce` method to emit IR that generates result tuples. Consider compiling the physical plan $\sigma_p(scan(T))$. `σ.produce` calls `scan.produce`, which emits the `for` loop over T and calls `σ.consume` to consume the tuples of the scan. Then, `σ.consume` inlines the selection over tuples of T in the for loop. The final emitted pseudocode will be:

```
for t in relation
  if p(t)
    <σ's parent logic>(t)
```

Operators such as hash aggregation and the building side of hash joins are called *pipeline breakers* because the entire pipeline (i.e., all operators up to the pipeline breaker) needs to materialize its end result before the next pipeline can start operating. Each pipeline defines a separate code block in the final emitted program code, where each code block is a separate `for` loop (similar to the example above).

Finally, it is important to note that interpretation-based execution models can also enable the tight integration principle by introducing new physical operators similar to the ones we presented in Section 3.2. Designing such a system is an interesting future direction.

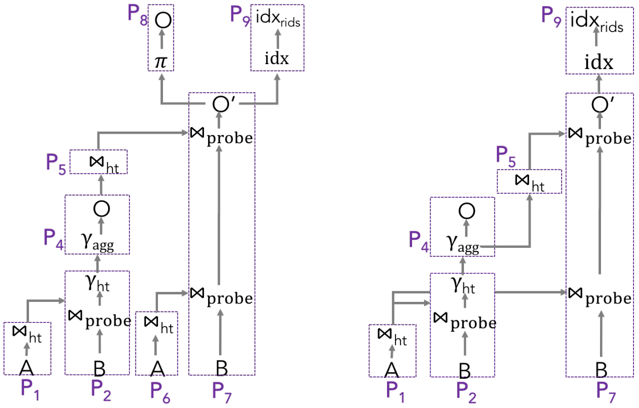
B. TUNING

Having provided a basic background on the query compilation and execution model of Smoke, in this section we present low-level optimization details that we enabled for lineage capture techniques. We start with logical approaches, then discuss Smoke-based optimizations, and conclude with physical approaches.

Tuning logical techniques. In our preliminary experiments we used Perm and GProm as they are implemented over Postgres 8.3 and a commercial database system, respectively. Unfortunately, both systems exhibit increased lineage capture overhead for reasons intrinsic to the underlying DBMSs which are not related to the principals behind the logical rewrite rules of Perm and GProm.

In particular, we observed increased lineage capture overhead due to (a) the added complexities from the transactional processing layers of full-fledged database systems [39] and (b) no flexibility in reusing data structures in the same physical query plan; reusing data structures is important for lineage capture, as we also noted for our techniques in Section 3. For these reasons, we implemented the rewrite rules of Perm and GProm in Smoke because (a) it does not incur the overhead of the transactional processing layers of a full-fledged database and (b) reuses data structures within the same query plan—hence, enables a fair comparison of only the principals behind Smoke and logical lineage capture techniques having fixed the underlying execution engine. Finally, we note that our implementation of logical alternatives in Smoke outperforms the available versions of Perm and GProm by two orders of magnitude because it avoids these caveats. For instance, Logic-Rid and Logic-Tup for 1m tuples and 1000 groups in our group-by aggregation microbenchmark take $< 200ms$ while Perm and GProm take $25s - 45s$. (depending on optimization knobs.)

To better explain the optimizations that we implemented for logical approaches, consider the following query which has the same structure with the queries that we used for our multi-operator experiments: $\gamma_{a,b,count(1)}(A \bowtie B)$, where $a \in \mathrm{sch}(A)$ and $b \in \mathrm{sch}(B)$; $\mathrm{sch}(X)$ denotes the schema (i.e., set



(a) Non-Optimized

(b) Optimized

Figure 16: Physical plans derived for logical approaches for the lineage capture of $\gamma_{a,b,count(1)}(A \bowtie B)$: (a) non-optimized and (b) optimized plan generated by Smoke. Boxes P_1-P_9 correspond to individual pipelines per the query compilation model. The index i of a pipeline P_i denotes the order of execution in the compiled plan.

of attributes) of relation X. Furthermore, let the join between A and B to be a pk-fk natural join.

Perm's re-rewrite rule for this query results in the following query: $\rho_{a/a',b/b'}(\gamma_{a,b,count(1)}(A \bowtie B)) \underset{a=a' \wedge b=b'}{\bowtie} (A \bowtie B)$; $\rho_{p/p'}$ denotes the relational rename operation of the attribute p to p'. Figure 16 shows the corresponding non-optimized and optimized physical plans that Smoke produces for this query. Similar non-optimized physical plans, with the same lack of optimizations, are produced by Perm and GProm. Next, we describe the set of optimizations to derive the optimized plan.

First, Perm joins the output of the aggregation with the join $A \bowtie B$ on the same attributes that the hash table was built on for the group-by aggregation. This allows us to reuse the hash table to join the output of the join $A \bowtie B$ with the aggregation result. As such, the built of the hash table at pipeline P_5 can be eliminated as we can reuse directly the hash table that was built during pipeline P_2. Second, consider the materialization of the output at pipeline P_8 and the materialization of the same output at P_7. This is the exact output of the base query. P_8 is more costly than P_7 due to the projection on a, possibly, large O' relation. Hence, we can eliminate P_8 and provide the output result O by executing P_4. Finally, note that the two natural joins between the tables A and B are the same. Hence, if we materialize the join output there is no need to perform again the join. In our experiments we found that joins are costly to materialize. As an alternative we re-used the hash table built for the inner table A and probed only the outer table B which resulted in the best performance in our experiments with the TPC-H queries. In general, for all n-way joins we materialized the last hash table and probed only the rightmost table in the left-deep plans that Smoke produces.

Note that the set of optimizations that we presented above can be enabled by any DBMS. This means that there is no intrinsic problem with the re-write rules of Perm and GProm, beyond the ones that we described in Section 2 and experimentally showed in our experiments. Rather, DBMSs, on top of which logical systems are built, need to perform low-level optimizations of physical plans. This is why we did not compare against systems that do not enable these optimizations and directly optimized logical alternatives within Smoke.

Tuning Smoke techniques. In Section 3.3 we noted that Smoke specifically optimizes SPJA queries with pk-fk joins. Note that the set of optimizations presented above are also enabled by Smoke for Smoke-D. The difference is that Smoke-D does not need to materialize O' as in P_7. Instead, it combines together P_7 and P_9 to build directly lineage indexes and always outperforms the logical approaches even with the optimizations because it avoids storing the denormalized representation that logical approaches perform. Furthermore, in our experiments we discussed that Smoke-I oupterforms Smoke-D in TPC-H queries that we experimented with. For these TPC-H queries, Smoke-I results in annotating intermediate hash-tables with rids as part of lineage capture for joins and the final aggregation operator uses these rids to perform the Inject method. Hence, Smoke-I avoids the expensive join of Smoke-D and outperforms it.

Tuning physical techniques. Similarly to logical techniques, we have also implemented Phys-Mem and Phys-Bdb within Smoke. More specifically, for a compiled query plan as generated by Lazy, we instrument it with virtual function calls to emit lineage (i.e., <output, input> rids). Then, assume that we want to capture backward lineage. For one-to-many relations between output and input, Phys-Mem probes a hash table on the output rid. Each entry in the hash table keeps a pointer to an rid index that we use to append the input rid. For one-to-one relations, we use an rid list where we append the input rid. In contrast, Phys-Bdb, adds the <output, input> rids in BerkeleyDB with the key being the output rid. (For forward lineage the process is the same but we probe on the input rids and we append output rids.) For our experiments, we used BerkeleyDB 12.1 that we instructed to (a) be in-memory with cache size 0.5GiB (which is sufficient to avoid spooling to disk) and (b) use B-Tree as its internal indexing structure. Essentially, both approaches are similar to Smoke-I but Phys-Mem implements the capture logic of Smoke-I in a virtual function instead of having it inline in the compiled plan while Phys-Bdb uses the B-tree indexes of BerkeleyDB instead of the lineage indexes of Smoke.

C. VARIANTS OF TPC-H Q1

In Section 6.4 we evaluated the optimizations of Smoke on three variants of TPC-H Q1, namely, $Q1_a$; $Q1_b$; and $Q1_c$, and compared them with lazy lineage query approaches. Here, we give a more detailed description of these queries and their lazy alternatives. First, we give a brief background on lazy alternatives [21, 43].

Lazy lineage capture. Consider a base group-by aggregation query $O = \gamma_{g_1,\ldots,g_n,F}(I)$, where $g_1 \ldots g_n$ are group-by attributes and F is the set of aggregation functions. A backward lineage query $L_B(o \in O, I)$ for a given output $o \in O$ is equivalent to $O_{\text{lazy}} = \sigma_{o.g_1=I.g_1 \wedge \ldots \wedge o.g_n=I.g_n}(I)$. Furthermore, if the base query O includes selections, these selections need to be added in the selection clause of O_{lazy}. Essentially, instead of accessing lineage through indexes as in the eager approaches of Smoke, the lazy approaches access lineage with selection scans on base relations. Similar are the equivalence rules for backward (or forward) lineage queries on top of joins, selections, projections, or general workflows [21, 43].

Now, recall that TPC-H Q1 is specified as shown below (Smoke's hash-based evaluation precludes sorting and ORDER BY clauses are omitted):

```
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 < '1998-12-01'
GROUP BY
  l_returnflag,
  l_linestatus
```

The output of $Q1$ is four groups derived from combinations of l_returnflag and l_linestatus (i.e., $\{(A,F),(N,O),(R,F),(N,F)\}$).

$Q1_a$ drills down from $Q1$ on the year and month of l_shipdate given an output group of $Q1$. Using with the backward lineage query construct this operation can be specified as follows:

```
Q1$_a$ = SELECT ...,
   extract(year from l_shipdate),
   extract(month from l_shipdate)
FROM $L_b(O_i \subseteq$ Q1 ,lineitem)
GROUP BY
  extract(year from l_shipdate),
  extract(month from l_shipdate)
```

For the eager approach of Smoke, $Q1_a$ is evaluated using only the lineage indexes that we construct during the execution of $Q1$. As a result, we do not need to add the selections specified in $Q1$ in $Q1_a$ because the backward lineage will retrieve tuples of lineitem that satisfy these selections.

Now, regarding the lazy approach to evaluate $Q1_a$, recall that a backward lineage for a group-by aggregation query $O = \gamma_{g_1,\ldots,g_n,F}(I)$ can be specified lazily as $\sigma_{o.g_1=I.g_1 \wedge \ldots \wedge o.g_n=I.g_n}(I)$. By applying this rule on $Q1_a$ we derive its lazy alternative:

```
Q1$_{a-lazy}$ =
   SELECT ...,
     extract(year from l_shipdate),
     extract(month from l_shipdate)
   FROM lineitem
   WHERE
     l_shipdate < '1998-12-01' and
     l_linestatus = ?  and
     l_returnflag = ?
   GROUP BY
     extract(year from l_shipdate),
     extract(month from l_shipdate)
```

Given an output group of $Q1$ we can parameterize the selections above and evaluate $Q1_a$ using a selection scan on lineitem (as opposed to the indexed scan using our Smoke techniques). Finally, note that in our experiments we considered backward lineage queries to be specified for only one output group of $Q1$. This avoids disjunctive selections on `l_linestatus` and `l_shipdate` (e.g., change the single selection `l_linestatus = ?` above to `l_linestatus IN (...)`), since we only need to backward trace to tuples that contribute only to one group. However, in the general case, lazy approaches need to consider these expensive disjunctions. In contrast, eager approaches have access to tuples of each group through the lineage indexes and do not require to re-calculate groups with expensive selections.

Following the logic of $Q1_a$ above we similarly derived the eager and lazy alternatives of $Q1_b$ and $Q1_c$. For completeness, we list them below without further discussion.

```
Q1$_b$ =
   SELECT ... FROM L$_b$(Q1'$\subseteq$Q1,lineitem)
   WHERE l_shipinstruct = ? and
         l_shipmode = ?
   GROUP BY ...

Q1$_{b-lazy}$ =
   SELECT ... FROM lineitem
   WHERE l_shipdate < '1998-12-01'
         l_shipinstruct = ? and l_shipmode = ? and
         l_linestatus = ? and l_returnflag = ?
   GROUP BY ...

Q1$_c$ =
   SELECT ... FROM L$_b$($Q1_b{}' \subseteq Q1_b$,lineitem)
   GROUP BY ..., l_tax

Q1$_{c-lazy}$ =
   SELECT ... FROM lineitem
   WHERE l_shipdate < '1998-12-01' and
         l_shipinstruct = ? and l_shipmode = ? and
         l_linestatus = ? and l_returnflag = ? and
         extract(year from l_shipdate) = ?
         extract(month from l_shipdate) = ?
    GROUP BY ..., l_tax
```

D. CROSSFILTERING USING LINEAGE

In Section 6.5.1, we introduced two techniques (i.e., BT and BT+FT) for crossfiltering using lineage that we compared with the Lazy approach. In this section, we give a more detailed technical discussion. Figure 17 shows the proposed physical plans (i.e., BT and BT+FT) for crossfiltering alongside the naive approach (i.e., Lazy) and drives our discussion.

Consider a set of queries $\mathcal{Q}$ = {Q$_x$| Q$_x$ = `SELECT` G$_x$, F$_x$(J$_x$) `FROM T GROUP BY` G$_x$} for which we seek to support crossfiltering functionality,

Lazy. During the execution of each Q_x, Lazy simply executes the group-by aggregations without capturing lineage. Then, given a selection of a subset of outputs of $Q_{\text{brushed}} \in \mathcal{Q}$ (i.e., $Q'_{\text{brushed}} \subseteq Q_{\text{brushed}}$), Lazy supports crossfiltering by updating each $Q_x \in \mathcal{Q} \setminus \{Q_{\text{brushed}}\}$ as follows:

```
Q$_x$' =SELECT G$_x$, F$_x$(J$_x$)
    FROM T
    WHERE $\bigwedge_{o \in Q'_{\text{brushed}}} o.G_b = T.G_b$
    GROUP BY G$_x$
```

Essentially, Lazy supports crossfiltering by performing lazy lineage capture to identify the partitions of the base relation that contributed to the selected outputs. Finally, to evaluate the set of all updated Q'_x, Lazy does not execute each update separately. Rather it uses a shared selection scan of the input relation with the selection being $\bigwedge_{o \in Q'_{\text{brushed}}} o.G_b = T.G_b$ to avoid multiple, expensive selection scans of the base relation.

During the execution of initial group-by aggregations, BT and BT+FT perform lineage capture in order to to use lineage indexes for crossfiltering.

BT. As we noted above, Lazy supports crossfiltering with lazy lineage capture. This means that Q'_x above is equivalent to the following query.

```
Q$_x$' =SELECT G$_x$,F$_x$(J$_x$)
     FROM backward_trace($Q'_{\text{brushed}} \subseteq Q_{\text{brushed}}$,T)
     GROUP BY G$_x$
```

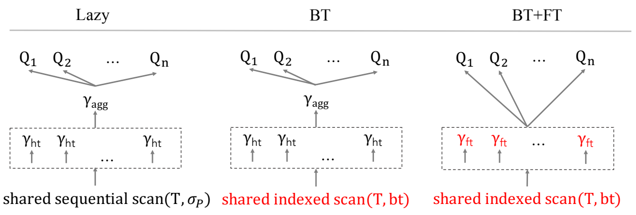

Lazy
BT
BT+FT
Q_1 Q_2 ... Q_n
γ_{agg}
γ_{ht} γ_{ht} ... γ_{ht}
γ_{ft} γ_{ft} ... γ_{ft}
shared sequential scan(T, σ_P)
shared indexed scan(T, bt)
shared indexed scan(T, bt)

Figure 17: Crossfilter evaluation techniques without using data cubes: (a) Lazy re-evaluates the group-by aggregation queries with a shared selection scan on the base table, (b) BT uses an index scan on the rids of the backward lineage index of Q'_{brushed}, (c) BT+FT performs updates using the fordward indexes that connect each tuple in the base table to each output of aggregation query.

Since we have built a backward index $\text{bt}_{Q_{\text{brushed}}}$, during the execution of Q_{brushed}, then Q'_x can be evaluated using an indexed-scan on the base table T using only the rid arrays of $\text{bt}_{Q'_{\text{brushed}}}$ of $\text{bt}_{Q_{\text{brushed}}}$ that correspond to the selected output records Q'_{brushed}. Essentially, with this approach we can avoid the selection scans of Lazy with indexed scans from lineage. Similarly to Lazy, BT uses a shared scan but this time a shared indexed-scan based on $\text{bt}_{Q'_{\text{brushed}}}$.

BT+FT. Finally, note that BT still needs to perform group-by aggregations which is a very costly operation due to the construction of hash tables. Instead, recall the notion of a forward lineage index for a group-by aggregation query: Each tuple in the input is associated with a group in the output. So, forward indexes provide a perfect hashing between tuples in the base table and group-by aggregation results that we have already calculated (i.e., the initial views). Hence, instead of constructing hash tables, BT+FT uses the forward indexes as perfect hash tables and performs crossfiltering as follows:

```
Input:  bw[][]        // backward index for $Q'_{brushed}$
        fw[][]        // forward indexes from each tuple
                      // to groups of each initial
                      // group-by aggregate
        $Q_1,...,Q_n$        // outputs of initial views
Output: $Q'_1,...,Q'_n$       // crossfiltered views
Init $Q'_1,...,Q'_n$ using $Q_1,...,Q_n$
for i = 0 to bw.size()
  for j = 0 to bw[i].size()
    for z = 0 to n
      agg_update($Q'_z$[fw[$Q_z$][bw[i][j]]])
remove_non_affected_groups($Q'_1,...,Q'_n$)
```

Listing 1: Crossfilter using BT+FT.

`agg_update()` in the listing above simply updates the aggregation (e.g., for `COUNT(*)`, `agg_update` is simply Q'_z`[fw[`Q_z`][bw[i][j]]]++`). Finally, note the `remove_non_affected_groups` function at the end of the listing above. This function loops over the groups of each updated group-by and removes the groups that were not affected by the group-by. In the case of `COUNT(*)` this is simply the groups that have a zero count. For other aggregates, like `SUM`, we need to track which groups were updated within the `agg_update` functions. However, in most interactive visualizations it is more important to maintain the groups even if the have a zero count (or a zero sum). In that case the `remove_non_affected_groups` can simply by ignored and the `agg_update` can perform the update without updating a state of what groups were updated. Our experiments in Section 6.5.1 report the latency of BT+FT including the time for this operation.

E. LINEAGE SEMANTICS IN SMOKE

In Section 2, we noted that Smoke uses transformation provenance semantics that can allow us to encode several novel provenance semantics at the will of end-developers. Here, we give a brief discussion on how one can go about encoding new semantics in Smoke.

Consider the following example query and database: `SELECT COUNT(*), A.cname, B.pname FROM A.cid = B.id GROUP BY A.cname, B.pname`.

	cid	cname
a_1	1	Bob
a_2	2	Alice

	oid	cid	pname	date
b_1	1	1	iPhone	12/25
b_2	2	1	iPhone	12/25
b_3	3	2	XBox	12/25

The output of this query is the following:

	COUNT(*)	A.cname	B.pname
o_1	1	Bob	iPhone
o_2	2	Alice	xBox

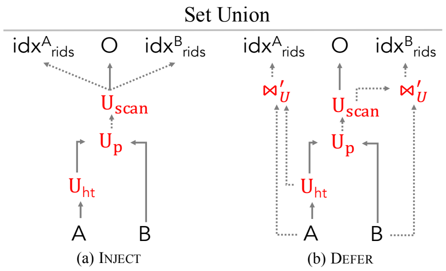
According to Smoke's provenance semantics the backward index for o_1 with respect to table A contains the tuple rid a_1 *twice*. This is important because in this way Smoke can encode multiple provenance semantics. The why-provenance of o_1 is $\{(a_1, b_1), (a_1, b_2)\}$ and to answer why-provenance queries Smoke simply concatenates the backward index rids: rids at the same position in the backward indexes for A and B correspond to the why-provenance witnesses. Now, the which-provenance of o_1 is $\{a_1, b_1, b_2\}$ which Smoke can derive by performing a set union of the backward indexes. Finally, the how-provenance of o_1 is $a_1 \cdot (b_1 + b_2)$ which Smoke can derive by set unioning and concatenating the backward rid indexes, similarly to the operations for which- and why-provenance.

Now, note that all these operations to derive different provenance semantics are lineage consuming queries whose logic we can push down as workload-aware optimizations similar to the ones in Section 4. In which case Smoke operates as a which-, why-, or how-provenance system for the case illustrated above. However, note that depending on the provenance semiring that how-provenance captures, the lineage consuming logic can be different and expressing semirings as lineage consuming queries is an open question.

These observations, along with the focus of Smoke to capture forward lineage, highlight that Smoke provides a general architecture for novel provenance semantics.

F. INSTRUMENTATION ALGEBRA

In Section 3 we presented the physical algebra of Smoke for lineage capture of projection, selections, hash-based group-by



(a) Inject (b) Defer

Figure 18: Inject and Defer plans for set union. Dotted arrows are only necessary for lineage capture.

```
Input:  A, B
Output: O,
        a_fw[A.size()], b_fw[B.size()] // forward indexes
        a_bw[][], b_bw[][]            // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\cup_{ht}$: Build phase
  h = hash(A[i].uattrs)
  if(!ht[h]) ht[h]={init_state(A[i].uattrs),
                    a_rids=[], b_rids=[]}
  ht[h].a_rids.insert(i)

for i = 0 to B.size()   // $\cup_{p}$: Probe/Append phase
  h = hash(B[i].uattrs)
  if(!ht[h]) ht[h]={init_state(B[i].uattrs),
                    , a_rids=[], b_rids=[]}
  ht[h].b_rids.insert(i)

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for (state, a_rids, b_rids) in ht   // $\cup_{scan}$: Scan phase
  O[++oid] = create_output_record(state)
  a_bw[oid] = a_rids
  for rid in a_rids
    a_fw[rid] = oid
  b_bw[oid] = b_rids
  for rid in b_rids
    b_fw[rid] = oid
```

Listing 2: Inject lineage capture for set union $A\overset{S}{\bigcup}_{\text{uattrs}}B$.

aggregations, and hash-based equi-joins. In this section, we extend this algebra for bag and set union, intersection, and difference as well as nested-loop θ-joins and cross products.

F.1 Set Union

Set union between two relations A and B (i.e., $A\overset{S}{\bigcup}_{\text{uattrs}}B$, where S denotes set union and uattrs the attributes from A and B to union on) are implemented in a hash-based way with consecutive appends to a hash table: Initially, the operator $\cup_{ht}$ builds a hash table using the relation A with the key being the attributes of the union (i.e., uattrs). Then, $\cup_p$ probes the hash table constructed by $\cup_{ht}$ on the union attributes using relation B. If an entry does not already exist for the union attributes, $\cup_p$ appends a new entry in the hash table with the union attributes. Essentially, $\cup_{ht}$ and $\cup_b$ are the same operator, that probe and append tuples in a hash table. The only difference is that $\cup_{ht}$ takes as input an empty hash table while $\cup_p$ takes as input a pre-built hash table. Finally, $\cup_{\text{scan}}$ scans the hash table and constructs the output. Next, we discuss Defer and Inject lineage capture approaches for set union; Figure 18 shows their corresponding physical plans.

Inject: Listing 2 illustrates the Inject lineage capture of Smoke for set union. Similarly to group-by aggregation, Inject rewrites $\cup_{ht}$ to append, besides the union attributes, two arrays a_rids and b_rids that track which tuples from A and B, respectively, contribute to the hash table entry. During $\cup_{ht}$ we populate a_rids and during $\cup_p$ we populate b_rids. Finally, $\cup_{\text{scan}}$ outputs the result and the lineage indexes.

Defer: Listing 3 illustrates the Defer lineage capture of Smoke for set union. Similarly to group-by aggregation, Defer rewrites $\cup_{ht}$ and $\cup_p$ to append an oid to each hash table entry, initially set to -1, besides the union attributes. Then, $\cup_{\text{scan}}$ outputs the set union result and assigns the correct oid to each hash table entry. To construct the lineage indexes $\bowtie'_{\cup}$ takes as input the relation A and probes the previously constructed hash table to find the oid and properly construct the lineage indexes between the output and input relation A.

```
Input:  A, B
Output: O,
        a_fw[A.size()], b_fw[B.size()] // forward indexes
        a_bw[][], b_bw[][]            // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\cup_{ht}$: Build phase
  h = hash(A[i].uattrs)
  if(!ht[h]) ht[h]={init_state(A[i].uattrs),
                    oid=-1}

for i = 0 to B.size()   // $\cup_{p}$: Probe/Append phase
  h = hash(B[i].uattrs)
  if(!ht[h]) ht[h]={init_state(B[i].uattrs),
                    , oid=-1}

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for h in ht   // $\cup_{scan}$: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid
for i=0 to A.size() // $\bowtie'_{\cup}$: Lineage capture for A
  h = hash(A[i].uattrs)
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
for i=0 to B.size() // $\bowtie'_{\cup}$: Lineage capture for B
  h = hash(B[i].uattrs)
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
```

Listing 3: Defer lineage capture for set union $A\overset{S}{\bigcup}_{\text{uattrs}}B$.

Similar is the process for $\bowtie'_{\cup}$ when taking as input the B to construct lineage indexes between the output and relation B.

Further optimizations: An optimization, for both Inject and Defer approaches, is that there is no need to wait to append the right relation B to the hash table to construct the lineage indexes for the relation A. This is because the intermediate hash table built for A suffices for the lineage index construction for A. For Defer, in particular, this also means that the join $\bowtie_U$ for A will not need to probe a hash table that keeps not all entries for A but also B. However, this also means that Defer needs to block the output construction until after the $\bowtie_U$ for A has been executed, which is a counter-argument to the Defer paradigm (i.e., lineage is constructed without blocking the query execution). To balance this effect we could keep a copy of the intermediate hash table for A and use only that for lineage construction for the A relation at the cost of copying which could be substantial. Smoke does not yet support the copy construction but it does support blocking the set union for the lineage construction.

F.2 Bag Union

Lineage capture for bag union is simpler than lineage capture for set union. Since for bag union we only concatenate the two input relations, what we only need to maintain is the *rid* of where one relation ends and the other relation begins in the output of the union. More generally, for bag union of k relations we need $k-1$ such rids. Using these indexes it is sufficient to answer both backward and forward lineage queries. Note, however that this lineage capture relies on the fact that the input relation is a base relation stored in the database. For multi-operator plans the input to the union could be an intermediate relation for which we need to perform lineage capture. For instance, for a query $\sigma_\theta(A)\bigcup B$, we need to perform lineage capture for the selection on A.

```
Input:  A, B
Output: O,
        a_fw[A.size()], b_fw[B.size()] // forward indexes
        a_bw[][], b_bw[][]            // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\cap_{ht}$: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs), b_bit = 0
                 oid=-1}

for i = 0 to B.size()   // $\cap_p$: Probe/Append phase
  h = hash(B[i].iattrs)
  if(ht[h]) ht[h].b_bit=1

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for h in ht   // $\cap_{scan}$: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid
for i=0 to A.size() // $\bowtie'_{\cap}$: Lineage capture for $A$
  h = hash(A[i].iattrs)
  if(!h.b_bit) continue
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
for i=0 to B.size() // $\bowtie'_{\cap}$: Lineage capture for $B$
  h = hash(B[i].iattrs)
  if(!h) continue
  a_bw[ht[h].oid].insert(i)
  a_fw[i] = ht[h].oid
```

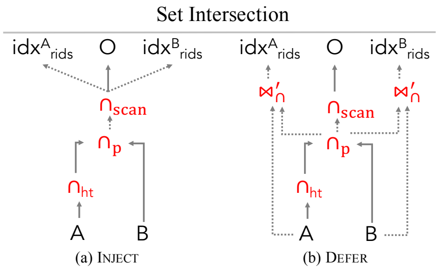
Listing 5: Defer lineage capture for set intersection $A \overset{S}{\bigcap}_{\text{iattrs}} B$.

```
Input:  A, B
Output: O,
        a_fw[A.size()], b_fw[B.size()] // forward indexes
        a_bw[][], b_bw[][]            // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\cap_{ht}$: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs),
                 a_rids=[], b_rids=[]}
  ht[h].a_rids.insert(i)

for i = 0 to B.size()   // $\cap_p$: Probe phase
  h = hash(B[i].iattrs)
  if(ht[h]) ht[h].b_rids.insert(i)

oid = -1
a_bw = int[ht.size()][]
b_bw = int[ht.size()][]
for (state, a_rids, b_rids) in ht   // $\cap_{scan}$: Scan phase
  if(b_rids.size()==0) continue;
  O[++oid] = create_output_record(state)
  a_bw[oid] = a_rids
  for rid in a_rids
    a_fw[rid] = oid
  b_bw[oid] = b_rids
  for rid in b_rids
    b_fw[rid] = oid
```

Listing 4: Inject lineage capture for set intersection $A \overset{S}{\bigcap}_{\text{iattrs}} B$.



(a) Inject (b) Defer

Figure 19: Inject and Defer plans for set intersection. Dotted arrows are only necessary for lineage capture.

F.3 Set Intersection

Set intersection in Smoke is broken into three operators. First, $\cap_{ht}$ builds a hash table on the outer relation A with the key being the attributes of the intersection. Each hash table entry, beyond the intersection attributes, also maintains a bit to indicate whether or not it has been matched with a tuple from the inner relation B. Then, $\cap_p$ probes the hash table and sets the bit if a match was found. Finally, $\cap_{\text{scan}}$ scans the hash table and emits the entries with the bit set to form the output.

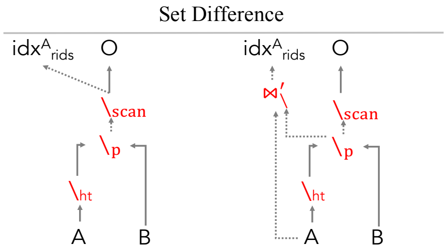
Linage capture for set intersection (see Figure 19) follows the logic of set union (see Figure 18). An important difference is that for the Inject approach, a_rids that we have kept for non-matched tuples will be discarded. If the fraction of tuples in the outer relation that appear in the intersection is small that could result in the Defer approach to be faster than Inject because it avoids the unnecessary writes in a_rids. Also, a slight difference from set intersection without lineage capture, is that Inject does not require a bit indicating whether a hash table entry has been matched with tuples from the outer relation because we maintain b_rids that provide this information. For completeness, Listings 4 and 5 show code snippets for Inject and Defer, respectively.

F.4 Bag Intersection

Bag intersection in Smoke follows the same logic as the set intersection. The only difference is that the hash table needs to maintain two more attributes per entry: (a) the number of tuples from the outter relation that are duplicates according to the intersection attributes, and (b) the number of matches with the inner relation. $\cap_{ht}$ adds a hash entry {A[i].iattrs, a_matches=1, b_matches=0} if there is no prior entry in the hash table for A[i].iattrs, or updates the matches of A (i.e., a_matches++) if there was an entry for A[i].iattrs. Then, $\cap_p$ probes the hash tables with the tuples from the inner relation B and updates the b_matches. Finally, $\cap_{scan}$ scans the hash table and outputs each entry a_matches·b_matches times to provide an output with the correct bag intersection semantics.

Inject: Lineage capture for bag intersection under Inject semantics is straightforward. Instead of keeping a_matches and b_matches we maintain two arrays of rids (a_rids and b_rids) from where the matches have originated. As such, a_matches = a_rids.size() and b_matches=b_rids.size(). Hence, $\cap_{scan}$ can still provide an output with the correct bag intersection semantics. Moreover, $\cap_{scan}$ can provide backward and forward indexes using these rids. Note, however, that while set intersection has 1-to-N backward lineage, bag intersection has 1-to-1.

Defer: Lineage capture for bag intersection under Defer follows the logic of Defer for set intersection. Besides a_matches and b_matches, each hash entry maintains an output rid oid of the first tuple in the output for this hash entry. Note that the output will contain tuples related to this hash entry at rids [oid, oid+a_matches·b_matches]. Now, the trick is that $\bowtie'_{\cap}$ need to happen in order first with the A relation and then with B, and for every match we should increase the oid. For completeness, Listing 6 provides the corresponding code snippet for Defer.



(a) Inject (b) Defer

Figure 20: Inject and Defer plans for set difference. Red operators are lineage-aware. Dotted arrows are only necessary for lineage capture.

```
Input:  A, B
Output: O,
        a_fw[A.size()], b_fw[B.size()] // forward indexes
        a_bw[][], b_bw[][]             // backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\cap_{ht}$: Build phase
  h = hash(A[i].iattrs)
  if(!ht[h]) ht[h]={init_state(A[i].iattrs),
                     a_matches=1, b_matches=0,
                    oid=-1}
  else ht[h].a_matches++

cnt=0
for i = 0 to B.size()   // $\cap_p$: Probe/Append phase
  h = hash(B[i].iattrs)
  if(ht[h])
    ht[h].b_matches++
    cnt+= a_matches

oid = -1
a_bw = int[cnt][]
b_bw = int[cnt][]
for h in ht   // $\cap_{scan}$: Scan phase
  O[++oid] = create_output_record(h.state)
  h.oid = oid
for i=0 to A.size() // $\bowtie'_{\cap}$: Lineage capture for $A$
  h = hash(A[i].iattrs)
  if(!h.b_matches) continue
  a_bw[ht[h].oid] = i
  a_fw[i] = ht[h].oid++
for i=0 to B.size() // $\bowtie'_{\cap}$: Lineage capture for $B$
  h = hash(B[i].iattrs)
  if(!h) continue
  a_bw[ht[h].oid] =i
  a_fw[i] = ht[h].oid++
```

Listing 6: Defer lineage capture for bag intersection $A\overset{B}{\bigcap}_{\text{iattrs}}B$.

F.5 Set and bag difference

Smoke implements set difference of two relations A and B (i.e., $A\overset{B/S}{\setminus}_{dattrs}B$) in a hash-based way similar to set intersection. The only differences are (a) we set the `b_bit` of each hash entry to 1 instead of 0 during the initial build and (b) when we probe the hash table with the inner relation we set the `b_bit` to 0 as opposed to 1. The final scan outputs only the hash entries with `b_bit=1` as these are the tuples that appear in the inner relation but do not appear in the outter relation.

Efficient lineage capture for set difference is non-trivial. By definition, the lineage for a tuple $o \in A \setminus B$ depends on (a) the set of tuples in A that it came from and (b) the whole inner relation B. Capturing forward indexes for the A tuples follows the lineage capture logic of set intersection and we omit further details. The problem with set difference is that each output depends on the whole outer relation B. Our experimental results show that lineage capture is meaningful when lineage has small cardinality. As such, if B is a base relation we do not capture lineage and for backward queries that require access to B we simply scan B. Now, if the input relation is an intermediate relation, then Smoke performs lineage capture during the execution of the operator whose output is the intermediate relation that is the outer relation to the set difference. Hence, for a backward query on the set difference we can access a base relation that is used to construct the intermediate relation through the backward index of the intermediate relation. More interestingly, a forward query from a tuple of a base relation, that is used in the construction of the intermediate relation that is input to the set difference, is the whole output *times* the amount of tuples it contributes to the intermediate relation. This is because each tuple in the intermediate relation contributes to all the tuples in the output of the set difference.

As such, Smoke captures lineage only for the A relation that follows the logic of lineage capture for the inner relation of set intersection. For completeness, Figure 20 illustrates the corresponding Inject and Defer physical plans.

F.6 θ-joins and Nested Loops

So far, we have proposed a physical algebra for hash-based implementations of equi-joins, group-by aggregations, unions, intersections, and differences. In this section we give a brief discussion for Inject lineage capture of nested-loop based implementations for θ-joins. Lineage capture with merge-sort approaches and lineage capture based on nested loops for the rest operators are obvious future work.

```
Input:  A, B
Output: O,
    a_fw[A.size()][], b_fw[B.size()][] // forward indexes
    a_bw[], b_bw[]              // backward indexes
oid=-1
for i = 0 to A.size()
  for j = 0 to B.size()
    if($\theta$(A[i], B[j]))
      O[++oid] = create_output_record(A[i], B[j])
      a_bw[oid] = i
      b_bw[oid] = j
      a_fw[i].insert(oid)
      b_fw[j].insert(oid)
```

Listing 7: Inject lineage capture for nested loop join $A \bowtie_\theta B$.

Inject: Listing 7 illustrates the lineage capture of Smoke for nested-loop θ-joins. For each combination of tuples from A and B that satisfy the θ condition the algorithm emits the record to construct the correct output. Since we write serially the output, we can also write serially the lineage indexes and maintain the alignment between each output record and their corresponding backward lineage index.

As an optimization, note that the backward index for the A relation can be condensed. All the output records due to $A[i]$ will be consecutive in the output. Hence, instead of keeping the rids for each output `a_fw[i].insert(oid)` we can simply store the rid of only the first one.

F.7 Cross product

Regarding cross product, Smoke does not perform lineage capture in the general case. Given an input tuple from the outer relation A with rid a we know that its forward lineage is $\{a, a+|B|, \ldots, a+(|A|-1)|B|\}$ due to the semantics of cross product. Similar is the series for the inner relation. Hence, whether we are given an input or output tuple we can directly infer the backward and lineage rids at runtime without a cost. If the input to cross product is intermediate relations, Smoke first captures lineage for operators that produce them.

F.8 Group-By Aggregations and Joins

Finally, we include code snippets for Defer and Inject joins and group-by aggregations that we presented in Section 3.2. Listings 8 and 9 illustrate the Inject and Defer approaches for group-by aggregation, respectively. Listing 10 illustrates the Inject approach of Smoke for joins, while Listing 11 shows the Defer approach and highlights its differences from the the Inject approach.

```
Input:  A
Output: O,
        fw[], bw[][]   // forward, backward index
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\gamma_{ht}$ Build phase
  h = hash(A[i].gbattr)
  if(!ht[h]) ht[h]={init_agg_state(), rids=[]}
  ht[h].state.update(A[i])
  ht[h].rids.insert(i)

fw = int[A.size()]
bw = int[ht.size()][]
oid = -1;
for (state, rids) in ht // $\gamma_{agg}$ Scan phase
  O[++oid] = create_output_record(state)
  bw[oid] = rids
  for rid in rids
    fw[rid] = oid
```

Listing 8: Inject lineage capture for γ_{ht} and γ_{agg}.

```
Input:  A
Output: O,
        fw[], bw[][]   // forward, backward index
Hash Table ht, Hash Function hash
for i = 0 to A.size()  // $\gamma_{ht}$ Build phase
  h = hash(A[i].gbattr)
  if(!ht[h]) ht[h]={init_agg_state(), oid :  -1}
  ht[h].state.update(A[i])


fw = int[A.size()]
bw = int[ht.size()][]
oid = -1;
for h in ht // $\gamma_{agg}$ Scan phase
  O[++oid] = create_output_record(h)
  h.oid = oid

for i=0 to A.size()
  h = hash(A[i].gbattr)
  bw[ht[h].oid].insert(i)
  fw[i] = ht[h].oid
```

Listing 9: Defer lineage capture using L_γ.

```
Input:  relations A, B;
Output: R                      // A $\bowtie_{A.a=B.b}$ B
        a_fw[][], b_fw[][]     // Forward indexes
        a_bw[],   b_bw[]       // Backward indexes
Hash Table ht, Hash Function hash
for i = 0 to A.size()          // Build Phase
  h = hash(A[i].a)
  if(!ht[h]) ht[h]={records=[], i_rids=[]}
  ht[h].records.insert(A[i])
  ht[h].i_rids.insert(i)

o = 0;
for i = 0 to B.size()          // Probe Phase
  h = hash(B[i].b)
  if(!(t = ht.probe(h))) continue;
  for j = 0 to t.i_rids.size()
     R[o] = (t.records[j], B[i])
     a_bw[o] = t.i_rids[j]
     b_bw[o] = i
     a_fw[j].insert(o)
     b_fw[i].insert(o++)
```

Listing 10: Inject lineage capture code for $A \bowtie_{A.a=B.b} B$.

```
...                    // Build Phase
  if(!ht[h])ht[h]={records=[],i_rids=[],o_rids=[]}
...
o = 0;
for i = 0 to B.size()        // Probe Phase
  h = hash(B[i].b)
  if(!(t = ht.probe(h))) continue;
  t.o_rids.insert(o)
  for j = 0 to t.i_rids.size()
    R[o] = (t.records[j], B[i])
    b_bw[o] = i
    b_fw[i].insert(o++)

a_bw = int[o]            // Build indexes for left relation
for h in ht
  s = 0
  for r in h.i_rids
    a_fw[r] = int[h.o_rids.size()]
    for o in h.o_rids
      a_fw[r].insert(o + s)
      a_bw[o+s] = r
    s++
```

Listing 11: Defer lineage capture for $A \bowtie_{A.a=B.b} B$.

G. MORE EXPERIMENTS

In this section, we include experiments that did not fit in the main body of the paper due to space limitations.

G.1 Microbenchmarks with Selection

This experiment uses the following base query: **SELECT** * **FROM** zipf **WHERE** v < ?, where the attribute v∈ $[0, 100]$ is drawn from a uniform distribution. Varying the parameter ? allows us to vary the query selectivity. Figure 21 reports the lineage capture costs for two relation sizes (1, 5 million), and varying the estimated query selectivity between 1% and 50%. We evaluate Smoke-I, as well as Smoke-I-EC, which estimates the query selectivity as $\frac{v}{100}$ and, in turn, uses the selectivity estimates to preallocate the lineage indexes.

Comparison of Smoke techniques for selection. Smoke-I introduces average overhead of 0.38× and 0.46×, for one and five million records across the varying selectivities. This is consistent with our finding that the techniques primarily vary by a constant per-tuple overhead. When using selectivity estimates, Smoke-I-EC reduces the average overhead to 0.14× and 0.15×, for the respective relation sizes. The reason that Smoke-I-EC fluctuates is that the selectivity estimates may be slightly incorrect. When estimates overestimate the true selectivity, it is typically fine, however if they underestimate then they lead to array resizing overheads.

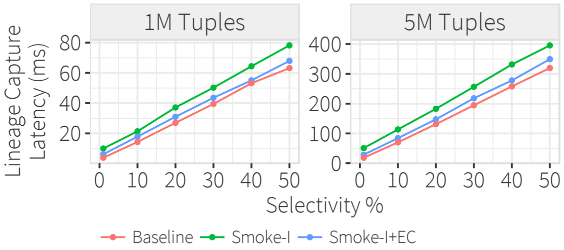


Figure 21: Instrumented selection latency with estimated predicate selectivity (Smoke-I-EC) and without (Smoke-I). We find that it is better to over estimate, than underestimate and incur resizing costs.

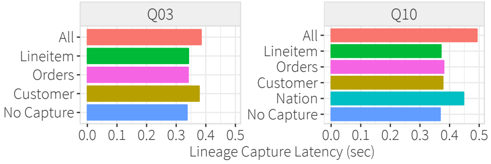


Figure 22: Lineage capture costs for different table pruning strategies. `ALL` refers to lineage capture for all tables. The lineage indexes for the left-most tables dominate the overhead (`Q3.Customer`, `Q10,Nation`)

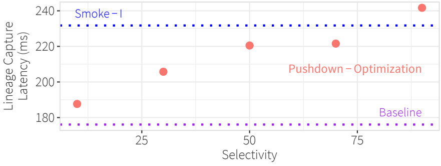


Figure 23: Lineage capture with selection push-down at varying selectivities of `l_tax < ?`. The crossover point between with and without pushdown is due to the additional cost of predicate evaluation before adding rids to the lineage indexes.

G.2 Workload-Aware Optimizations

This set of experiments evaluate the effects of instrumentation pruning and selection pushdown, which are designed to reduce lineage capture costs.

Pruning input relations. Figure 22 compares the latency of Q3 and Q10, which read three and four relations, respectively, under three sets of conditions: no lineage capture, lineage capture for all input relations (non-optimized Smoke-I), and Smoke-I-based lineage capture for a single input relation. We did not evaluate Q1, which is a single relation query, and Q12 shows the same findings. Although pruning input relations from lineage capture reduces the overall overhead, we find that the main overhead is due to the left-most tables in the join plans (`Customer` for Q3, `Nation` for Q10). It tends to be a smaller table, thus the fanout when joined with the other tables is high, and leads to more rid array reallocations. `Lineitem` incurs the lowest overhead because it is the right-most join relation, and its join is a primary-foreign key join. Our pk-fk join optimization uses an rid array rather than an rid index for the forward lineage index, which is much cheaper to populate.

Selection pushdown. To evaluate the impact of the selection pushdown optimization, we used Q1 as the base query, and ran the following lineage consuming query:

SELECT * **FROM** L_B(Q1, Lineitem) **WHERE** l_tax < ?

Figure 23 plots the average and standard deviation base query latency when assigning `?` to 5 distinct `l_tax` values, along with the cost of Smoke-I without selection pushdown, and Lazy. We find that the effectiveness of selection pushdown depends on the selectivity of the predicate. The overhead is linear with respect to the predicate selectivity, and there is a cross-over point with Smoke-I at high selectivities (> 75%), where the overhead of evaluating the predicate for every input record outweighs the benefits of building a smaller lineage index. We expect that increasing the predicate complexity (e.g., string comparisons, more predicate clauses) will likely shift the cross-over point towards lower selectivities. These results suggest the value of cost-based methods to choose between the two.

H. RELATED WORK

Lineage. Most related to our work are lineage subsystems for databases that model, capture, index, and query lineage information. In Section 2, we classified the different subsystems into logical [9, 21, 31, 32, 35, 66, 86] and physical [43,44,45,46,58,89]. , explained their differences, and discussed how Smoke avoids their drawbacks. Our experiments show how Smoke can both provide negligible lineage capture overhead, fast lineage query execution, and outperform state-of-the-art alternatives by multiple orders of magnitude.

Physical Database Design. The physical database design literature has long studied techniques to create redundant data structures (e.g., indexes and materialized views) and data layouts to minimize the expected execution cost of a possible future query workload [3,4,15,25,42,51,59,62,69,73,79,84]. Smoke is the first database engine to consider lineage as a type of information for physical design decisions (i.e., we showed how we can build online lineage indexes and push logic of lineage consuming queries into lineage capture to answer future queries equivalent to SQL queries). Also, Smoke does not simply push the physical database design costs into query execution; we both propose write-efficient data structures to minimize construction overheads and *overlap lineage index construction costs with query execution logic*.

Lineage Applications. A core motivation behind Smoke is to demonstrate how applications with hand-tuned implementations can leverage lineage systems to express their logic declaratively and enjoy out-of-the(-lineage)-box optimizations. Interactive data visualizations have long materialized specialized data structures—such as data cubes [8,49,57,85], indexes [55,71], or precomputed results [13]—offline in order to insure sub-150ms response times. In our experiments with crossfiltering we showed how our push-down optimizations can be used to construct such cubes. Moreover, we showed lineage-enabled techniques and workload-aware optimizations that can adequately address the cold-start problem of interactive visualizations [7] and the "Overview first, zoom and filter, and details on demand" interaction paradigm (by either generating indexes or materializing and partitioning results). Finally, we showed how lineage can express data profiling primitives, of core use across domains [63,76,83], and outperform hand-written implementations. These results provide evidence that Smoke *does not just provide declarative features for applications to express their logic but it can actually optimize applications holistically*.