an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Smoke: fine-grained lineage at interactive speed

SEPTEMBER 24, 2018

tags: Datastores

Smoke: fine-grained lineage at interactive speed (http://www.vldb.org/pvldb/vol11/p719-psallidas.pdf) Psallidas et al., *VLDB'18*

Data lineage connects the input and output data items of a computation. Given a set of output records, a *backward lineage query* selects a subset of the output records and asks "which input records contributed to these results?" A *forward lineage query* selects a subset of the input records and asks, "which output records depend on these inputs?". *Lineage-enabled systems* capture record-level relationships throughout a workflow and support lineage queries.

Data lineage is useful in lots of different applications; this paper uses as its main example interactive visualisation systems. This domain requires fast answers to queries and is typically dominated by hand-written implementations. Consider the two views in the figure below. When the user selects a set of marks in V_1, marks derived from the same records are highlighted in V_2 (*linked brushing*).

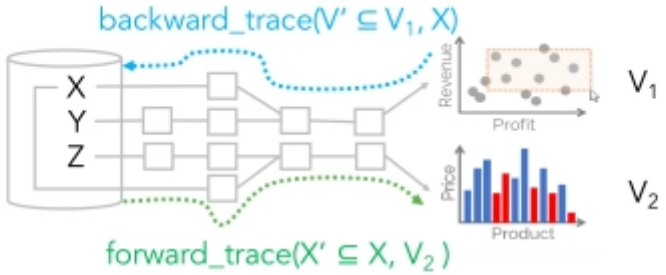

Figure 1: Two workflows generate visualization views V_1 and V_2. Then, a linked brushing interaction highlights in red marks in V_2 that share the same input records with selected marks of V_1. This interaction can be expressed as a backward query from selected circles in V_1 followed by a forward query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

A typical visualisation system implements this manually, but it can equally be viewed as a backward lineage query from the selection points in V_1, followed by a forward lineage query from the resulting input records to V_2.

(See '**Explaining outputs in modern data analytics' (https://blog.acolyer.org/2017/02/01/explaining-outputs-in-modern-data-analytics/)** which we looked at last year for an introduction to lineage and provenance principles. Chotia et al. use a *shadow* dataflow graph mapping the original computation but flowing in the opposite direction…).

Challenges with existing lineage capture systems

We have the usual space/time trade-offs to consider. We can slow down the *base query* in order to capture lineage information during query execution (and store that information somewhere). This speeds up answering lineage queries later on. Or we can keep base queries fast and lazily materialize lineage information later when lineage queries are asked (making them slower).

> *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.*

Smoke's four principles

Smoke employs four central design principles to try and pull off this trick.

1. Tight integration of lineage capture into query execution itself, using write-efficient data structures.
2. Where apriori knowledge of lineage queries is available (e.g., the set of explorations supported by a visualisation tool), this information is used to minimise the amount of lineage that needs to be materialized.
3. Again, if we know lineage queries in advance, and those queries are only interested in *aggregated* information then we can potentially materialize aggregate statistics as we process queries, and prune or re-partition lineage indices.
4. Wherever possible, data structures constructed during normal operation execution are augmented and reused for lineage purposes rather than introducing separated dedicated structures.

> *…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. 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.*

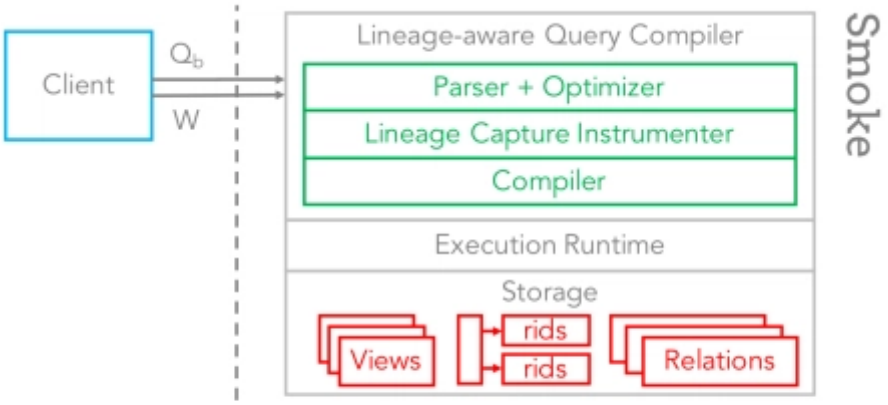

Figure 2: SMOKE is a query compilation engine that instruments physical query plans to capture lineage efficiently: Query execution generates lineage indexes that map input and output record ids (rids) as well as materialized views.

Lineage capture

Smoke uses read- and write-efficient index structures based on row ids to capture lineage information. 1-N relationships (between input and output tuples) are represented as inverted indexes. The index's *ith* entry corresponds to the *ith* output group, and points to a row id array containing the ids of all input records that belong to the group. 1-1 relationships between input and output are represented as a single array.

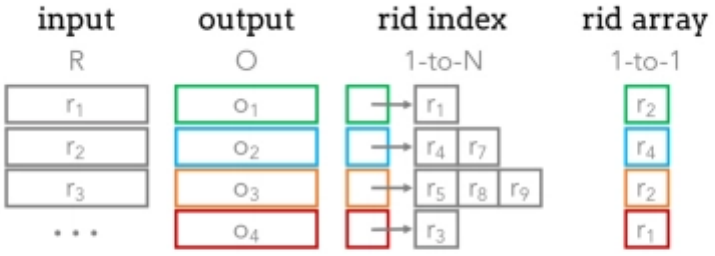

Figure 3: Lineage index representations: rid index for 1-to-N operators (e.g., GROUPBY); rid array for 1-to-1 operators (e.g., SELECT).

These indices are populated through a tight integration of lineage capture and relation operator logic to avoid additional API calls, and to facilitate co-optimisation. There are two basic strategies depending on the operator and circumstances: *defer* and *inject*.

Inject strategies incur the full cost of index generation during base query execution, whereas defer strategies defer (portions of) the lineage capture until after the operation execution.

Consider selection: both forward and backward lineage capture use row-id arrays, with the forward one pre-allocated based on the cardinality of the input relation. While iterating over the relation evaluating the predicate, the inject strategy adds two counters to track the row ids of the current input and output

and uses these to update the indices when an output row is emitted. There is no defer strategy for selection, because it is strictly inferior to inject.

As another example consider hash joins. Smoke will generate both backward row id arrays, and forward row-id indexes. Under the inject strategy a build phase augments each hash table entry with a row id array containing the input row ids for that entry's join key. The probe phase tracks the row id for each output record and populates the forward and backward indexes. One drawback of this strategy is that we might trigger multiple re-allocations (growing the size) of the forward indexes if an input record has many matches.

The *defer* strategy for hash joins takes advantage of the fact that we know the size of the forward indexes after the probe phase. The build phase maintains an additional output row id list, storing the first output record for each match (output records are emitted contiguously). After the probe phase, the forward and backward indexes can be pre-allocated and populated in a final scan of the hash table.

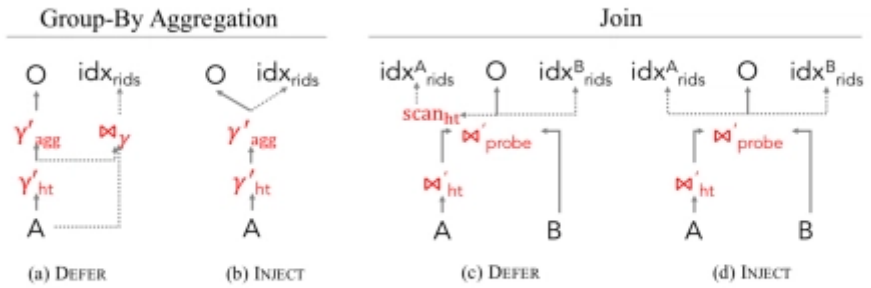

Figure 4: Inject and Defer plans for group-by aggregation and join. Dotted arrows are only necessary for lineage capture.

For *multi-operator* plans Smoke propagates lineage information through plan execution so that only a single set of lineage indexes connecting the input and final output relations are emitted. It also takes advantage of *pipelines* (that merge multiple operators into a single pipeline as part of normal query execution) to eliminate intermediate lineage materialization points where possible.

Workload-aware optimisations

When the set of lineage queries is known in advance, Smoke can go further than the baseline lineage capture describe above and also apply *instrumentation pruning* and *optimisation push-down*.

Instrumentation pruning disables lineage capture for lineage indexes that will not be used by the workload. Lineage queries that apply summarisation / aggregation before presenting results provide an opportunity to push-down optimisations. For example, selection push-down when using a static predicate, and partitioning of index arrays by predicate attributes for dynamic selections. Group-by aggregation can also be pushed down into lineage capture.

We observe that popular provenance semantics (e.g. which and why 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.

Evaluation

Smoke is compared to state-of-the-art logical and physical lineage capture and query approaches using a combination of microbenchmarks, TPC-H queries, and two real-world applications.

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

Table 1: Lineage capture techniques used in our evaluation.

Smoke's lineage capture techniques outperform both logical and physical approaches by up to two orders of magnitude. For example:

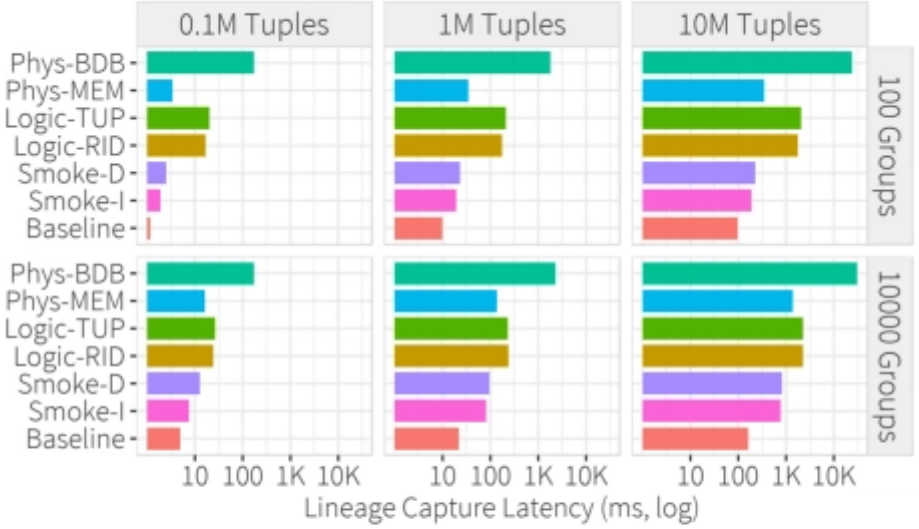

Figure 5: Comparison of lineage capture costs for the group-by aggregation operator for different relation cardinalities (columns) and number of distinct groups (rows). Smoke-I and Smoke-D slow down the non-instrumented Baseline the least as compared to alternative logical and physical capture methods.

Smoke also sped-up lineage query evaluation by multiple orders of magnitude, especially for low-selectivity lineage queries.

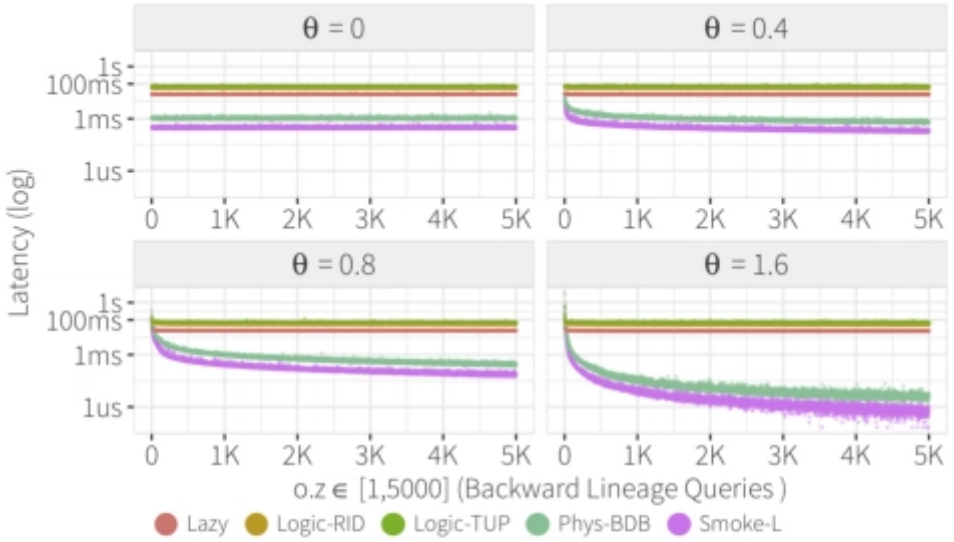

Figure 9: Lineage query latency for varying data skew (θ). Lazy has a fixed cost to scan the input relation and evaluates the simple selection predicate on the group-by key z=?. Logic-Rid and Logic-Tup performs the same selection but on annotated output relations. Smoke-L is mainly around 1*ms*, and 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.

The two real-world applications used in the evaluation were Crossfilter visualisation and data profiling with UGuide. The results show that :

> Lineage can express many real-world tasks from visualisation and data profiling, that are currently implemented by hand in ad-hoc ways, and
> the lineage capture mechanism is fast enough to avoid sacrificing performance vs the hand implementations, and in many cases may even perform better.

> *Smoke illustrates that it is possible to both capture lineage with low overhead and enable fast lineage query performance… Our capture techniques and workload-aware optimization make Smoke well-suited for online; adaptive; and offline physical database design settings.*

from → Uncategorized