In-memory Incremental Maintenance of Provenance Sketches

Pengyuan Li
Illinois Institute of Technology
USA
pli26@hawk.iit.edu

Boris Glavic
University of Illinois Chicago
USA
bglavic@uic.edu

Dieter Gawlick
Oracle Corporation
USA
dieter.gawlick@oracle.com

Vasudha Krishnaswamy
Oracle Corporation
USA
vasudha.krishnaswamy@oracle.com

Zhen Hua Liu
Oracle Corporation
USA
zhen.liu@oracle.com

Danica Porobic
Oracle Corporation
USA
danica.porobic@oracle.com

Xing Niu
Oracle Corporation
USA
xing.niu@oracle.com

Abstract

Provenance-based data skipping [38] compactly over-approximates the provenance of a query using so-called provenance sketches and utilizes such sketches to speed-up the execution of subsequent queries by skipping irrelevant data. However, a sketch captured at some time in the past may become stale if the data has been updated subsequently. Thus, there is a need to maintain provenance sketches. In this work, we introduce **I**n-Memory incremental **M**aintenance of **P**rovenance sketches (IMP), a framework for maintaining sketches incrementally under updates. At the core of IMP is an incremental query engine for data annotated with sketches that exploits the coarse-grained nature of sketches to enable novel optimizations. We experimentally demonstrate that IMP significantly reduces the cost of sketch maintenance, thereby enabling the use of provenance sketches for a broad range of workloads that involve updates.

Keywords

Provenance-based Data Skipping, Incremental Maintenance, Updates, Provenance, Dynamic Relevance Analysis

1 Introduction

Database engines take advantage of physical design such as index structures, zone maps [33] and partitioning to prune irrelevant data as early as possible during query evaluation. In order to prune data, database systems need to determine statically (at query compile time) what data is needed to answer a query and which physical design artifacts to use to skip irrelevant data. For instance, to answer a query with a `WHERE` clause condition `A = 3` filtering the rows of a table `R`, the optimizer may decide to use an index on `A` to filter out rows that do not fulfill the condition. However, as was demonstrated in [38], for important classes of queries like queries involving top-k and aggregation with `HAVING`, it is not possible to determine *statically* what data is needed, motivating the use of *dynamic relevance analysis* techniques that determine during query execution what data is relevant to answer a query. In [38] we introduced such a dynamic relevance analysis technique called provenance-based data skipping (PDBS). In PDBS, we encode what data is relevant for a query as a so-called *provenance sketch*. Given a range-partition of a table accessed by a query, a provenance sketch records which fragments of the partition contain provenance. That is, provenance sketches compactly encode an over-approximation of the provenance of a query. [38] presents safety conditions that ensure a sketch is *sufficient*, i.e., evaluating the query over the data represented by the sketch is guaranteed to produce the same result as evaluating the query over the full database. Thus, sketches are used to speed up queries by filtering data not in the sketch.

EXAMPLE 1.1. *Consider the database shown in Fig. 1 and query Q_{Top} that returns products whose total sale volume is greater than \$5000. The provenance of the single result tuple $(Apple, 5074)$ are the two tuples (tuples s_3 and s_4 shown with purple background), as the group for Apple is the only group that fulfills the* `HAVING` *clause. To create a provenance sketch for this query, we select a range-partition of the* `sales` *table that optionally may correspond to the physical storage layout of this table. For instance, we may choose to partition on attribute* `price` *based on ranges ϕ_{price}:*

$$\{\rho_1 = [1, 600], \rho_2 = [601, 1000], \rho_3 = [1001, 1500], \rho_4 = [1501, 10000]\}$$

In Fig. 1, we show for each tuple the fragment f_i it belongs to. Here fragment f_i corresponds to range ρ_i. Two fragments (f_3 and f_4 highlighted in red) contain provenance and, thus, the provenance sketches for Q_{Top} wrt. $F_{\mathbf{sales},price}$ is $\mathcal{P} = \{\rho_3, \rho_4\}$. Evaluating the query over the sketch's data is guaranteed to produce the same result as evaluation on the full database.[1]

As demonstrated in [38], provenance-based data skipping can significantly improve query performance — we pay upfront for creating sketches for some of the queries of a workload and then amortize this cost by using sketches to answer future queries by skipping irrelevant data. To create, or *capture*, a sketch for a query Q we execute an instrumented version of Q. Similarly, to use a sketch for a query Q, this query is instrumented to filter out data that does not belong to the sketch. For instance, consider the sketch for Q_{Top} from Ex. 1.1 containing two ranges $\rho_3 = [1001, 1500]$ and $\rho_4 = [1501, 10000]$. To skip irrelevant data, we create a disjunction of conditions testing that each tuple passing

EDBT '26, Tampere (Finland)

[1] In general, this is not the case for non-monotone queries. The safety check from [38] can be used to test whether a particular partition for a table is safe for a query. The partition used here fulfills this condition.

Q_{Top}

```
SELECT brand, SUM(price * numSold) AS rev
FROM sales
GROUP BY brand
HAVING SUM(price * numSold) > 5000
```

brand	rev
Apple	5074

sales

	sid	brand	productName	price	numSold	
s_1	1	Lenovo	ThinkPad T14s Gen 2	349	1	f_1
s_2	2	Lenovo	ThinkPad T14s Gen 2	449	2	f_1
s_3	3	Apple	MacBook Air 13-inch	1199	1	f_3
s_4	4	Apple	MacBook Pro 14-inch	3875	1	f_4
s_5	5	Dell	Dell XPS 13 Laptop	1345	1	f_3
s_6	6	HP	HP ProBook 450 G9	999	4	f_2
s_7	7	HP	HP ProBook 550 G9	899	1	f_2

Figure 1: Example query and relevant subsets of the database.

the `WHERE` clause belongs to the sketch, i.e., has a price within ρ_3 or ρ_4:[2]

```
WHERE (price BETWEEN 1001 AND 1500)
   OR (price BETWEEN 1501 AND 10000)
```

PDBS enables databases to exploit physical design for new classes of queries, significantly improving the performance of aggregation queries with `HAVING` and top-k queries [38] and, more generally, any query where only a fraction of the database is relevant for answering the query. For instance, for a top-k query only tuples contributing to a result tuple in the top-k are relevant, but which tuples are in the top-k can only be determined at runtime. Counterexamples include queries with selection conditions with low selectivity for which the database can effectively filter the data without PDBS.

However, just like materialized views, a sketch captured in the past may no longer correctly reflect what data is needed (has become *stale*) when the database is updated. The sketch then has to be maintained to be valid for the current version of the database.

EXAMPLE 1.2 (STALE SKETCHES). *Continuing with our running example, consider the effect of inserting a new tuple*

s_8 = `(8, HP, HP ProBook 650 G10, 1299, 1)`

into relation ***sales****. Running Q_{Top} over the updated table returns a second result tuple* `(HP, 6194)` *as the total revenue for HP is now above the threshold specified in the* `HAVING` *clause. For the updated database, the three tuples for HP also belong to the provenance. Thus, the sketch has become stale as it is missing the range ρ_2 which contains these tuples. Evaluating Q_{Top} over the outdated sketch leads to an incorrect result that misses the group for HP.*

Consider a partition F of a table R accessed by a query Q. We use $Q_{R,F}$ to denote the *capture query* for Q and F, generated using the rewrite rules from [38]. Such a query propagates coarse-grained provenance information and ultimately returns a sketch. A straightforward approach to maintain sketches under updates is *full maintenance* which means that we rerun the sketch's *capture query* $Q_{R,F}$ to regenerate the sketch. Typically, $Q_{R,F}$ is more expensive than Q. Thus, frequent execution of capture queries is not feasible. Alternatively, we could employ incremental view maintenance (IVM) techniques [23, 26] to maintain $Q_{R,F}$. However, capture queries use specialized data types and functions to efficiently implement common operations related to sketches. For instance, we use bitvectors to encode sketches compactly and utilize optimized (aggregate) functions and comparison operators for this encoding. To give two concrete examples, a function implementing binary search over the set of ranges for a sketch is used to determine which fragment an input tuple belongs to and an aggregation function that computes the bitwise-or of multiple bitvectors is used to implement the union of a set of partial sketches. To the best of our knowledge these operations are not supported by state-of-the-art IVM frameworks. Furthermore, sketches are compact over-approximations of the provenance of a query that are *sound*: evaluating the query over the sketch yields the same result as evaluating it over the full database. It is often possible to further over-approximate the sketch, trading improved maintenance performance for increased sketch size. Existing IVM methods do not support such trade-offs as they have to ensure that incremental maintenance yields the same result as full maintenance.

In this work, we study the problem of maintaining sketches under updates such that a sketch created in the past can be updated to be valid for the current state of the database. Towards this goal we develop an incremental maintenance framework for sketches that respects the approximate nature of sketches, has specialized data structures for representing data annotated with sketches and maintenance rules tailored for sketch-annotated data.

We start by introducing a data model where each row is associated with a sketch and then develop incremental maintenance rules for operators over such annotated relations. We then present an implementation of these rules in an in-memory incremental engine called IMP (**I**ncremental **M**aintenance of **P**rovenance Sketches). The input to this engine is a set of annotated delta tuples (tuples that are inserted / deleted) that we extract from a backend DBMS. To maintain a sketch created by a capture query $Q_{R,F}$ at some point in the past, we extract the delta between the current version of the database and the database instance at the original time of capture (or the last time we maintained the sketch) and then feed this delta as input to our incremental engine to compute a delta for the sketch. IMP outsources some of the computation to the backend database. This is in particular useful for operations like joins where deltas from one side of the join have to be joined with the full table on the other side similar to the delta rule $\Delta R \bowtie S$ used in standard incremental view maintenance. Additionally, we present several optimizations of our approach: (i) filtering deltas determined by the database to prune delta tuples that are guaranteed to not affect the result of incremental maintenance, (ii) filtering deltas for joins using bloom filters and (iii) reducing the size of the state for top-k operators. IMP is effective for any query that benefits from sketches, e.g., queries with `HAVING`, as long as the cost of maintaining sketches is amortized by using sketches for answering queries.

In summary, we present IMP, the first incremental engine for maintaining provenance sketches. Our main contributions are:

- We develop incremental versions of relational algebra operators for sketch-annotated data.
- We implement these operators in IMP, an in-memory engine for incremental sketch maintenance. IMP enables PDBS for any DBMS by acting as a middleware between the user and the database that manages and maintains sketches.
- We experimentally compare IMP against full maintenance and against a baseline that does not use PDBS using TPC-H, real world datasets and synthetic data. IMP outperforms full maintenance, often by several orders of magnitude. PDBS with IMP

[2]Note that the conditions for adjacent ranges in a sketch can be merged. Thus, the actual instrumentation would be `WHERE price BETWEEN 1001 AND 10000`.

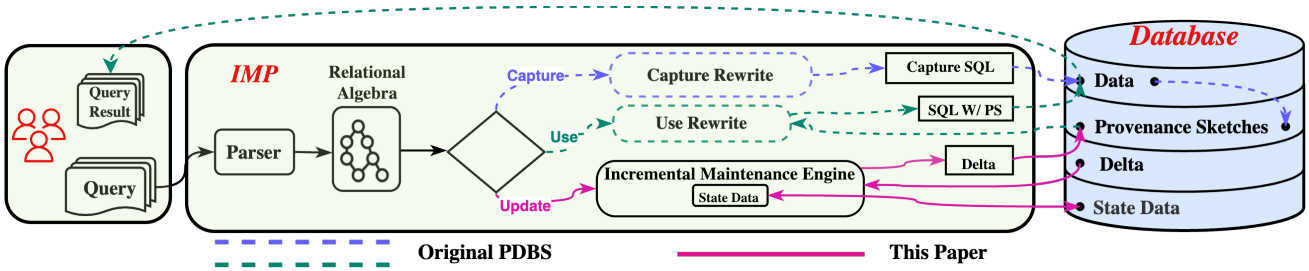


Figure 2: IMP manages a set of sketches. For each incoming query, IMP determines whether to (i) capture a new sketch, (ii) use an existing non-stale sketch, or (iii) incrementally maintain a stale sketch and then utilize the updated sketch to answer the query.

significantly improves the performance of mixed workloads including both queries and updates.

The remainder of this paper is organized as follows: Sec. 2 presents an overview of IMP. We discuss related work in Sec. 3. We formally define incremental maintenance of sketches and introduce our annotated data model in Sec. 4. In Sec. 5, we introduce incremental sketch maintenance rules for relational operators and prove their correctness in Sec. 5.4. We discuss IMP's implementation in Sec. 6, present experiments in Sec. 7, and conclude in Sec. 8.

2 Overview of IMP

Fig. 2 shows a overview of IMP that operates as a middleware between the user and a DBMS. We highlight parts of the system that utilize techniques from [38]. The dashed blue pipeline rewrites queries to capture sketches while the dashed green pipeline rewrites queries to use sketches. Users send SQL queries and updates to IMP that are parsed using IMP's parser and translated into an intermediate representation (relational algebra with update operations). The system stores a set of provenance sketches in the database. For each sketch we store the sketch itself, the query it was captured for, the current state of incremental operators for this query, and the database version it was last maintained at or first captured at for sketches that have not been maintained yet. As sketches are small (100s of bytes), we treat sketches as immutable and retain some or all past versions of a sketch. This has the advantage that it avoids write conflicts (for updating the sketch) between concurrent transactions that need to access different versions of the sketch. We assume that the DBMS uses snapshot isolation and we can use snapshot identifiers used by the database internally to identify versions of sketches and of the database. For systems that use other concurrency control mechanisms, IMP can maintain version identifiers. Furthermore, the system can persist the state that it maintains for its incremental operators in the database. This enables the system to continue incremental maintenance from a consistent state, e.g., when the database is restarted, or when we are running out of memory and need to evict the operator states for a query. IMP enables PDBS for workloads with updates on top of any SQL databases.

IMP supports multiple incremental maintenance strategies. Under *eager* maintenance, the system incrementally maintains each sketch that may be affected by the update (based on which tables are referenced by the sketch's query) by processing the update, retrieving the delta from the database, and running the incremental maintenance. Eager maintenance can be configured to batch updates. If the operator states for a sketch's query are not currently in memory, they will be fetched from the database. The updates to the sketches determined by incremental maintenance are then directly applied. Under *lazy* maintenance, the system passes updates directly to the database. When a sketch is needed to answer a query, this triggers maintenance for the sketch. For that, IMP fetches the delta between the version of the database at the time of the last maintenance for the sketch and the current database state and incrementally maintains the sketch. The result is a sketch that is valid as of the current state of the database. More advanced strategies can be designed on top of these two primitives, e.g., triggering eager maintenance during times of low resource usage or eagerly maintaining sketches for queries with strict response time requirements to avoid slowing down such queries when maintenance is run for a large batch of updates.

For queries sent by the user, IMP first determines whether there exists a sketch that can be used to answer the query Q. For that, it applies the mechanism from [38] to determine whether a sketch captured for a query Q' in the past can be safely used to answer Q. If such a sketch $\mathcal{P}$ exists, we determine whether $\mathcal{P}$ is stale. If that is the case, then IMP incrementally maintains the sketch to $\mathcal{P}'$ (solid red pipeline). Afterwards, the query Q is instrumented to filter input data based on sketch $\mathcal{P}'$ and then the instrumented query is sent to the database and its results are forwarded to the user (the dashed green pipeline using techniques from [38]). If no existing provenance sketch can be used to answer Q, then IMP creates a capture query for Q and evaluates this query to create a new sketch $\mathcal{P}$ (dashed blue pipeline [38]). This sketch is then used to answer Q (dashed green pipeline [38]). IMP is an in-memory engine, exploiting the fact that sketches are small and that deltas and state required for incremental maintenance are typically small enough to fit into main memory or can be split into multiple batches if this is not that case.

3 Related Work

Provenance. Provenance can be captured by annotating data and propagating these annotations using relational queries or by extending the database system [25, 39, 40]. Systems like GProM [7], Perm [19], Smoke [42], Smoked Duck [34], Links [16], ProvSQL [43] and DBNotes [8] capture provenance for SQL queries. In [38], we introduced provenance-based data skipping (PBDS). The approach captures sketches over-approximating the provenance of a query and utilizes these sketches to speed-up subsequent queries. We present the first approach for maintaining sketches under updates, thus, enabling efficient PBDS for databases that are subject to updates.

Incremental View Maintenance (IVM). View maintenance has been studied extensively [9, 11, 17, 23, 27, 44]. [22, 45] gives an overview of many techniques and applications of view maintenance. Early work on view maintenance, e.g., [9, 11], used set semantics. This was later expanded to bag semantics (e.g., [13, 20]). We consider bag semantics. Materialization has been studied for

$D(\Delta D)$	a database (a delta database)
$\mathscr{D}(\Delta\mathscr{D})$	an annotated database (an annotated delta database)
$R(\mathscr{R}, \Delta\mathscr{R})$	a relation (an annotated relation, an annotated delta relation)
$\langle t, \mathcal{P}\rangle$	an annotated tuple: a tuple t associated with its provenance sketch $\mathcal{P}$
ρ, ϕ, Φ	a range, set of ranges for partitioning relation R (database D)
$\mathcal{S}$	state of an incremental relational algebra operator
$\mathcal{P}$	a provenance sketch
$\mathcal{I}$	an incremental maintenance procedure
Q	a query

Figure 3: Glossary

Datalog as well [21, 23, 35]. Incremental maintenance algorithms for iterative computations have been studied in [1, 10, 36, 37]. [26] proposed higher-order IVM. [46] maintains aggregate views in temporal databases. [41] proposes a general mechanism for aggregation functions. [2, 48] studied automated tuning of materialized views and indexes in databases. As mentioned before, existing view maintenance techniques can not be directly applied for provenance sketches maintenance, since [38] uses specialized data types and functions to efficiently handle sketches during capture, which are not supported in state-of-the-art IVM systems. Furthermore, classical IVMs solutions have no notion of over-approximating query results and, thus, can not trade sketch accuracy for performance. Several strategies have been studied for maintaining views eagerly and lazily. For instance, [14] presented algorithms for deferred maintenance and [9, 12, 23] studied immediate view maintenance. Our approach supports both cases: immediately maintaining sketches after each update or sketches can be updated lazily when needed.

Maintaining Provenance. [47] presents a system for maintenance of provenance in a distributed Datalog engine. In contrast to our work, [47] is concerned with efficient distributed computation and storage for provenance. Provenance maintenance has to deal with large provenance annotations that are generated by complex queries involving joins and operations like aggregation that compute a small number of result tuples based on a large number of inputs. [47] addresses this problem by splitting the storage of provenance annotations across intermediate query results requiring recursive reconstruction at query time. In contrast, provenance sketches are small and their size is determined upfront based on the partitioning that is used. Because of this and because of their coarse-grained nature, sketches enable new optimizations, including trading accuracy for performance.

4 Background and Problem Definition

In this section we introduce necessary background and introduce notation used in the following sections. Let $\mathbb{U}$ be a domain of values. An instance R of an n-ary relation schema $\text{SCH}(R) = (a_1, \ldots, a_n)$ is a function $\mathbb{U}^n \to \mathbb{N}$ mapping tuples to their multiplicity. We use $\{\!\{\cdot\}\!\}$ to denote bags and $t^n \in R$ to denote that tuple t exists with multiplicity n in relation R, i.e., $R(t) = n$. A database D is a set of relations R_1 to R_m. The schema of a database $\text{SCH}(D)$ is the set of relation schemas $\text{SCH}(R_i)$ for $i \in [1, m]$. Fig. 4 shows the bag semantics relational algebra used in this work. We use $\text{SCH}(Q)$ to denote the schema of the query Q and $Q(D)$ to denote the result of evaluating query Q over database D. Selection $\sigma_\theta(R)$ returns all tuples from relation R which satisfy the condition θ. Projection $\Pi_A(R)$ projects all input tuples on a list of projection expressions. Here, A denotes a list of expressions with potential renaming (denoted by $e \to a$) and $t.A$ denotes applying these expressions to a tuple t. For example, $a + b \to c$ denotes renaming the result of $a + b$ as c. $R \times S$ is the cross product for bags. For convenience we also define join $R \bowtie_\theta S$ and natural join $R \bowtie S$ in the usual way. Aggregation $\gamma_{f(a);G}(R)$ groups tuples according to their values in attributes G and computes the aggregation function f over the bag of values of attribute a for each group. We also allow the attribute storing $f(a)$ to be named explicitly, e.g., $\gamma_{f(a)\to x;G}(R)$, renames $f(a)$ as x. Duplicate removal $\delta(R)$ removes duplicates (definable using aggregation). The top-k operator $\tau_{k,O}(R)$ returns the first k tuples from the relation R sorted on order-by attributes O. We use $<_O$ to denote the order induced by O. The position of a tuple in R ordered on O is denoted by $\mathbf{pos}(t, R, O)$ and defined as: $\mathbf{pos}(t, R, O) = \sum_{t' <_O t} R(t')$. Fig. 3 shows an overview of the notations used in this work.

$$\sigma_\theta(R) = \{\!\{t^n | t^n \in R \wedge t \models \theta\}\!\} \quad \Pi_A(R) = \{\!\{t^n | n = \sum_{u.A=t} R(u)\}\!\}$$

$$\delta(R) = \{\!\{t^1 | t \in R\}\!\} \quad R \times S = \{\!\{(t \circ s)^{n*m} | t^n \in R \wedge s^m \in S\}\!\}$$

$$\gamma_{f(a);G}(R) = \{\!\{(t.G, f(G_t))^1 | t \in R\}\!\}$$
$$G_t = \{(t_1.a)^n | {t_1}^n \in R \wedge t_1.G = t.G\}$$

$$\tau_{k,O}(R) = \{\!\{t^m \mid \mathbf{pos}(t, R, O) < k \wedge m = min(R(t), k - \mathbf{pos}(t, R, O))\}\!\}$$

Figure 4: Bag Relational Algebra

4.1 Range-based Provenance Sketches

We use provenance sketches to concisely represent a superset of the provenance of a query (a sufficient subset of the input) based on horizontal partitions of the input relations of the query.

4.1.1 Range Partitioning. Given a set of intervals over the domains of a set of *partition attributes* $A \subset \text{SCH}(R)$, *range partitioning* determines membership of tuples to fragments based on their A values. For simplicity, we define partitioning for a single attribute a, but all of our techniques also apply when $|A| > 1$.

DEFINITION 4.1 (RANGE PARTITION). *Consider a relation R and $a \in \text{SCH}(R)$. Let $\mathbb{D}(a)$ denote the domain of a and $\phi = \{\rho_1, \ldots, \rho_n\}$ be a set of intervals $[l, u] \subseteq \mathbb{D}(a)$ such that $\bigcup_{i=0}^{n} \rho_i = \mathbb{D}(a)$ and $\rho_i \cap \rho_j = \emptyset$ for $i \neq j$. The* range-partition *of R on a according to ϕ denoted as $F_{\phi,a}(R)$ is defined as:*

$$F_{\phi,a}(R) = \{R_{\rho_1}, \ldots, R_{\rho_n}\} \quad \textbf{where} \quad R_\rho = \{\!\{t^n \mid t^n \in R \wedge t.a \in \rho\}\!\}$$

We will use F instead of $F_{\phi,a}$ if ϕ and a are clear from the context and f, f', f_i, etc. to denote fragments. We also extend range partitioning to databases. For a database $D = \{R_1, \ldots, R_n\}$, we use Φ to denote a set of range - attribute pairs $\{(\phi_1, a_1), \ldots, (\phi_n, a_n)\}$ such that F_{ϕ_i,a_i} is a partition for R_i. Relations R_i that do not have a sketch can be modeled by setting

$$\rho_i = \{[min(\mathbb{D}(a_i)), \max(\mathbb{D}(a_i))]\}$$

,a single range covering all domain values.

4.1.2 Provenance Sketches. Consider a database D, query Q, and a range partition of D using ranges Φ. We use $P(Q, D) \subseteq D$ to denote the provenance of Q wrt. D. For the purpose of PDBS, any provenance model that represents the provenance of Q as a subset of D can be used as long as the model guarantees *sufficiency*[3] [18]: $Q(P(Q, D)) = Q(D)$. A provenance sketch $\mathcal{P}$ for Q according to Φ is a subset of the ranges ϕ_i for each $\phi_i \in \Phi$ such that the fragments corresponding to the ranges in $\mathcal{P}$ fully cover Q's provenance within each R_i in D, i.e.,$P(Q, D) \cap R_i$. We will write $\rho \in \Phi$ to

[3]Note that our notion of sufficiency aligns with the one from [24] which differs slightly from the one used in [18] that is defined for a single result tuple of Q.

denote that $\rho \in \phi_i$ for some $\phi_i \in \Phi$ and D_ρ for ρ from ϕ_i to denote the subsets of the database where all relations are empty except for R_i which is set to $R_{i,\rho}$, the fragment for ρ. We use $\mathcal{P}_\Phi(D, \Phi, Q) \subseteq \Phi$ to denote the set of ranges whose fragments overlap with the provenance $P(Q, D)$:

$$\mathcal{P}_\Phi(D, \Phi, Q) = \{\rho \mid \rho \in \phi_i \wedge \exists t \in P(Q, D) : t \in R_{i,\rho}\}$$

Definition 4.2 (Provenance Sketch). *Let Q be a query, D a database, R a relation accessed by Q, and Φ a partition of D. We call a subset $\mathcal{P}$ of Φ a **provenance sketch** iff $\mathcal{P} \supseteq \mathcal{P}_\Phi(D, \Phi, Q)$. A sketch is **accurate** if $\mathcal{P} = \mathcal{P}_\Phi(D, \Phi, Q)$. The **instance** $D_\mathcal{P}$ of $\mathcal{P}$ is defined as $D_\mathcal{P} = \bigcup_{\rho \in \mathcal{P}} D_\rho$. A sketch is **safe** if $Q(D_\mathcal{P}) = Q(D)$.*

Consider the database consisting of a single relation (`sales`) from our running example shown in Fig. 1. According to the partition $\Phi = \{(\phi_{price}, price)\}$, the accurate provenance sketch $\mathcal{P}$ for the query Q_{Top} according to Φ consists of the set of ranges $\{\rho_3, \rho_4\}$ (the two tuples in the provenance of this query highlighted in Fig. 1 belong to the fragments f_3 and f_4 corresponding to these ranges). The instance $D_\mathcal{P}$, i.e., the data covered by the sketch, consists of all tuples contained in fragments f_3 and f_4 which are: $\{s_3, s_4, s_5\}$. This sketch is safe. We use the method from [38] to determine for an attribute a and query Q whether a sketch build on any partition of R on a will be safe.

4.2 Updates, Histories, and Deltas

For the purpose of incremental maintenance we are interested in the difference between database states. Given two databases D_1 and D_2 we define the *delta* between D_1 and D_2 to be the symmetric difference between D_1 and D_2 where tuples t that have to be inserted into D_1 to generate D_2 are tagged as $\mathbb{A}t$ and tuples that have to be deleted to derive D_2 from D_1 are tagged as $\mathbb{D}t$:

$$\Delta(D_1, D_2) = \{\!\{\mathbb{D}t \mid t \in D_1 - D_2\}\!\} \cup \{\!\{\mathbb{A}t \mid t \in D_2 - D_1\}\!\}$$

For a given delta ΔD, we use $\mathbb{D}D$ ($\mathbb{A}D$) to denote $\{\!\{\mathbb{D}t \mid \mathbb{D}t \in \Delta D\}\!\}$ ($\{\!\{\mathbb{A}t \mid \mathbb{A}t \in \Delta D\}\!\}$). We use $D \uplus \Delta D$ to denote *applying* delta ΔD to database D:

$$D \uplus \Delta D = D - \{\!\{\mathbb{D}t \mid \mathbb{D}t \in \Delta D\}\!\} \cup \{\!\{\mathbb{A}t \mid \mathbb{A}t \in \Delta D\}\!\}$$

Example 4.1. *Reconsider the insertion of tuple s_8 (also shown below) into **sales** as shown in Ex. 1.2.*

`s8 = (8, HP, HP ProBook 650 G10, 1299, 1)`

We get: $\Delta D = \{\!\{\mathbb{A}\ s_8\}\!\}$

We use the same delta notation for sketches, e.g., for two sketch versions $\mathcal{P}_1$ and $\mathcal{P}_2$, $\Delta\mathcal{P}$ is their delta if $\mathcal{P}_2 = \mathcal{P}_1 \uplus \Delta\mathcal{P}$, where $\uplus$ on sketches is defined as expected by inserting $\mathbb{A}\mathcal{P}$ and deleting $\mathbb{D}\mathcal{P}$.

4.3 Sketch-Annotated Databases And Deltas

Our incremental maintenance approach utilizes relations whose tuples are annotated with sketches. We define an incremental semantics for maintaining the results of operators over such annotated relations and demonstrate that this semantics correctly maintains sketches.

Definition 4.3 (Sketch Annotated Relation). *A sketch annotated relation $\mathcal{R}$ of arity m for a given set of ranges ϕ over the domain of some attribute $a \in \text{Sch}(R)$, is a bag of pairs $\langle t, \mathcal{P}\rangle$ such that t is an m-ary tuple and $\mathcal{P} \subseteq \phi$.*

We next define an operator $\textbf{annotate}(R, \Phi)$ that annotates each tuple with the singleton set containing the range its value in attribute a belongs to. This operator will be used to generate inputs for incremental relational algebra operators over annotated relations.

Definition 4.4 (Annotating Relations). *Given a relation R, attribute $a \in \text{Sch}(R)$ and ranges $\Phi = \{\ldots, (\phi, a), \ldots\}$, i.e., (ϕ, a) is the partition for R in Φ, the operator* **annotate** *returns a sketch-annotated relation $\mathcal{R}$ with the same schema as R:*

$$\textbf{annotate}(R, \Phi) = \{\!\{\langle t, \{\rho\}\rangle \mid t \in R \wedge t.a \in \rho \wedge \rho \in \phi\}\!\}$$

We define annotated deltas as deltas where each tuple is annotated using the **annotate** operator. Consider a delta ΔR between two versions R_1 and R_2 of relation R. Given ranges ϕ for attribute $a \in \text{Sch}(R)$, we define $\Delta\mathcal{R}$ as: $\Delta\mathcal{R} = \textbf{annotate}(\Delta R, \Phi)$. $\Delta\mathcal{R}$ contains all tuples from R that differ between R_1 and R_2 tagged with $\mathbb{A}$ or $\mathbb{D}$ depending on whether they got inserted or deleted. Each tuple t is annotated with the range $\rho \in \phi$ that $t.a$ belongs to. Analog we use $\mathcal{D}$ to denote the annotated version of database D and use $\Delta\mathcal{D}$ to denote the annotated version of delta database ΔD.

Example 4.2. *Continuing with Ex. 4.1, the annotated version of ΔD_2 according to ϕ_{price} is $\{\!\{\langle\mathbb{A}\ s_8, \{\rho_3\}\rangle\}\!\}$, because $s_8.price$ belongs to $\rho_3 = [1001, 1500] \in \phi_{price}$.*

4.4 Problem Definition

We are now ready to define *incremental maintenance procedures (IMs)* that maintain provenance sketches. An IM takes as input a query Q and an annotated delta $\Delta\mathcal{D}$ for the ranges Φ of a provenance sketch $\mathcal{P}$ and produces a delta $\Delta\mathcal{P}$ for the sketch. Note that we assume that all attributes used in Φ are *safe*. An attribute a is safe for a query Q if every sketch based on some range partition on a is safe. We use the safety test from [38] to determine safe attributes. IMs are allowed to store some state $\mathcal{S}$, e.g., information about groups produced by an aggregation operator, to allow for more efficient maintenance. Given the current state and $\Delta\mathcal{D}$, the IM should return a delta $\Delta\mathcal{P}$ for the sketch $\mathcal{P}$ and an updated state $\mathcal{S}'$ such that $\mathcal{P} \uplus \Delta\mathcal{P}$ over-approximates an accurate sketch for the updated database.

Definition 4.5 (Incremental Maintenance Procedure). *Given a query Q, a database D and a delta ΔD. Let $\mathcal{P}$ be a provenance sketch over D for Q wrt. some partition Φ. An* **incremental maintenance procedure** *$\mathcal{I}$ takes as input a state $\mathcal{S}$, the annotated delta $\Delta\mathcal{D}$, and returns an updated state $\mathcal{S}'$ and a provenance sketch delta $\Delta\mathcal{P}$:*

$$\mathcal{I}(Q, \Phi, \mathcal{S}, \Delta\mathcal{D}) = (\Delta\mathcal{P}, \mathcal{S}')$$

Let $\mathcal{P}[Q, \Phi, D]$ denote an accurate sketch for Q over D wrt. Φ. Niu et al. [38] demonstrated that any over-approximation of a safe sketch is also safe, i.e., evaluating the query over the over-approximated sketch yields the same result as evaluating the query over the full database. Thus, for a IM $\mathcal{I}$ to be correct, the following condition has to hold: for every sketch $\mathcal{P}$ that is valid for D and delta ΔD, $\mathcal{I}$, if provided with the state $\mathcal{S}$ for D and the annotated version $\Delta\mathcal{D}$ of ΔD, returns an over-approximation of the accurate sketch $\mathcal{P}[Q, \Phi, D \uplus \Delta D]$:

$$\mathcal{P}[Q, \Phi, D \uplus \Delta D] \subseteq \mathcal{P} \uplus \mathcal{I}(Q, \Phi, \mathcal{S}, \Delta\mathcal{D})$$

5 Incremental Annotated Semantics

We now introduce an IM that maintains sketches using annotated and incremental semantics for relational algebra operators. Each

operator takes as input an annotated delta produced by its inputs (or passed to the IM in case of the table access operator), updates its internal state, and outputs an annotated delta. Together, the states of all such incremental operators in a query make up the state of our IM. For an operator O (or query Q) we use $\mathcal{I}(O, \Phi, \Delta\mathcal{D}, \mathcal{S})$ ($\mathcal{I}(Q, \Phi, \Delta\mathcal{D}, \mathcal{S})$) to denote the result of evaluating O (Q) over the annotated delta $\Delta\mathcal{D}$ using the state $\mathcal{S}$. We will often drop $\mathcal{S}$ and Φ. Our IM evaluates a query Q expressed in relational algebra producing an updated state and outputting a delta where each row is annotated with a partial sketch delta. These partial sketch deltas are then combined into a final result $\Delta\mathcal{P}$.

Example 5.1. *Fig. 5 shows annotated tables $\mathcal{R}$ and $\mathcal{S}$, ranges ϕ_a and ϕ_c for attribute a (table R) and c (table S), the delta ΔR and the sketches before the delta has been applied: $\mathcal{P}_R$ and $\mathcal{P}_S$. Consider the following query over R and S:*

```
SELECT a, sum(c) as sc
FROM (SELECT a, b FROM R WHERE a > 3) JOIN S on (b = d)
GROUP BY a HAVING SUM(c) > 5
```

Fig. 5 (right table) shows each operator's output. We will further discuss these in the following when introducing the incremental semantics for individual operators. In this example, a new tuple is inserted into R resulting in sketch deltas $\Delta\mathcal{P}_R = \{\!\{ f_1 \}\!\}$ and $\Delta\mathcal{P}_S = \{\!\{ g_2 \}\!\}$. The tuple inserted into R results in the generation of a new group for the aggregation subquery which passes the `HAVING` *condition and in turn causes the two fragments from the tuple belonging to this group to be added to the sketches.*

5.1 Merging Sketch Deltas

Each incremental algebra operator returns an annotated relation where each tuple is associated with a sketch that is sufficient to produce it. To generate the sketch for a query Q we evaluate the query under our incremental annotated semantics to produce the tuples of $Q(D)$ each annotated with a partial sketch. We then combine these partial sketches into a sketch for Q. We now discuss the operator μ that implements this final merging step. To determine whether a change to the annotated query result will result in a change to the current sketch, this operator maintains as state a map $\mathcal{S} : \Phi \to \mathbb{N}$ that records for each range $\rho \in \Phi$ the number of result tuples for which ρ is in their sketch. If the counter for a fragment ρ reaches 0 (due to the deletion of tuples), then the fragment needs to be removed from the sketch. If the counter for a fragment ρ changes from 0 to a non-zero value, then the fragment now belongs to the sketch for the query (we have to add a delta inserting this fragment to the sketch).

$$\mathcal{I}(\mu(Q), \Delta\mathcal{D}, \mathcal{S}) = (\Delta\mathcal{P}, \mathcal{S}')$$

We first explain how $\mathcal{S}'$, the updated state for the operator, is computed and then explain how to compute $\Delta\mathcal{P}$ using $\mathcal{S}$. We define $\mathcal{S}'$ pointwise for a fragment ρ. Any newly inserted (deleted) tuple whose sketch includes ρ increases (decreases) the count for ρ. That is the total cardinality of such inserted tuples (of bag $\mathbb{\Delta}\ \mathcal{D}$ and $\Delta\ \mathcal{D}$, respectively) has to be added (subtracted) from the current count for ρ. Depending on the change of the count for ρ between $\mathcal{S}$ and $\mathcal{S}'$, the operator μ has to output a delta for $\mathcal{P}$. Specifically, if $\mathcal{S}[\rho] = 0 \neq \mathcal{S}'[\rho]$ then the fragment has to be inserted into the sketch and if $\mathcal{S}[\rho] \neq 0 = \mathcal{S}'[\rho]$ then the fragment was part of the sketch, but no longer contributes and needs to be removed.

$$\mathcal{S}'[\rho] = \mathcal{S}[\rho] + |\mathbb{\Delta}\ \mathcal{D}_\rho| - |\Delta\ \mathcal{D}_\rho|$$
$$\mathbb{\Delta}\ \mathcal{D}_\rho = \{\!\{ \mathbb{\Delta}\ \langle t, \mathcal{P}\rangle^n \mid \mathbb{\Delta}\ \langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \wedge \rho \in \mathcal{P} \}\!\}$$
$$\Delta\ \mathcal{D}_\rho = \{\!\{ \Delta\ \langle t, \mathcal{P}\rangle^n \mid \Delta\ \langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \wedge \rho \in \mathcal{P} \}\!\}$$
$$\Delta\mathcal{P} = \bigcup_{\rho:\, \mathcal{S}[\rho]=0 \wedge \mathcal{S}'[\rho]\neq 0} \{\mathbb{\Delta}\ \rho\} \cup \bigcup_{\rho:\, \mathcal{S}[\rho]\neq 0 \wedge \mathcal{S}'[\rho]=0} \{\Delta\ \rho\}$$

Example 5.2. *Reconsider our running example from Ex. 1.1 that partitions based on ϕ_{price}. Assume that there are two result tuples t_1 and t_2 of a query Q that have $\rho_2 = [601, 1000]$ in their sketch and one result tuple t_3 that has ρ_1 and ρ_2 in its sketch. Then the current sketch for the query is $\mathcal{P} = \{\rho_1, \rho_2\}$ and the state of μ is as shown below. If we are processing a delta $\Delta\ \langle t_3, \{\rho_1, \rho_2\}\rangle$ deleting tuple t_3, the updated counts $\mathcal{S}'$ are:*

$$\mathcal{S}[\rho_1] = 1 \quad \mathcal{S}[\rho_2] = 3 \qquad \mathcal{S}'[\rho_1] = 0 \quad \mathcal{S}'[\rho_2] = 2$$

As there is no longer any justification for ρ_1 to belong to the sketch (its count changed to 0), μ returns a delta: $\{\Delta\ \rho_1\}$

Consider the merge operator μ in Ex. 5.1. The state data before maintenance contains ranges f_2 and g_1. A single tuple annotated with f_1 and g_2 is added to the input of this operator. Both ranges were not present in $\mathcal{S}$ and, thus, in addition adding them to $\mathcal{S}'$ the merge operator returns a sketch delta $\mathbb{\Delta}\ \{f_1, g_2\}$.

5.2 Incremental Relational Algebra

5.2.1 Table Access Operator. The incremental version of the table access operator R returns the annotated delta $\Delta\mathcal{R}$ for R passed as part of $\Delta\mathcal{D}$ to the IM unmodified. This operator has no state.

$$\mathcal{I}(R, \Delta\mathcal{D}) = \Delta\mathcal{R}$$

Fig. 5 (top) shows the result of annotating relation ΔR from Ex. 5.1.

5.2.2 Projection. The projection operator does not maintain any state as each output tuple is produced independently from an input tuple if we consider multiple duplicates of the same tuple as separate tuples. For each annotated delta tuple $\Delta\langle t, \mathcal{P}\rangle$, we project t on the project expressions A and propagate $\mathcal{P}$ unmodified as $t.A$ in the result depends on the same input tuples as t.

$$\mathcal{I}(\Pi_A(Q), \Delta\mathcal{D}) = \{\!\{ \Delta\langle t.A, \mathcal{P}\rangle^n \mid \Delta\langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \}\!\}$$

5.2.3 Selection. The incremental selection operator is stateless and the sketch of an input tuple is sufficient for producing the same tuple in the output of selection. Thus, selection returns all input delta tuples that fulfill the selection condition unmodified and filters out all other delta tuples. In our running example (Fig. 5), the single input delta tuple fulfills the condition of selection $\sigma_{a>3}$.

$$\mathcal{I}(\sigma_\theta(Q), \Delta\mathcal{D}) = \{\!\{ \Delta\langle t, \mathcal{P}\rangle^n \mid \Delta\langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \wedge t \models \theta \}\!\}$$

5.2.4 Cross Product. The incremental version of a cross product (and join) $Q_1 \times Q_2$ combines three sets of deltas: (i) joining the delta of Q_1 with the current annotated state of Q_2 ($Q_2(\mathcal{D})$), (ii) joining the delta of the Q_2 with $Q_1(\mathcal{D})$, (iii) joining the deltas of Q_1 and Q_2. For (iii) there are four possible cases depending on which of the two delta tuples being joined is an insertion or a deletion. For two inserted tuples that join, the joined tuple $s \circ t$ is inserted into the result of the cross product. For two deleted tuples, we also have to insert the joined tuple $s \circ t$ into the result. For a deleted tuple joining an inserted tuple, we should delete the tuple $s \circ t$. The non-annotated version of these rules have been discussed in [14, 20, 26, 32]. We use ΔQ_i to denote $\mathcal{I}(Q_i, \Delta\mathcal{D})$ for $i \in \{1, 2\}$ below.

Table, Ranges and Delta

$\mathcal{R}$

a	b	$\mathcal{P}$
1	7	$\{f_1\}$
9	9	$\{f_2\}$

$\mathcal{S}$

c	d	$\mathcal{P}$
6	9	$\{g_1\}$
7	8	$\{g_2\}$

$\phi_a = \{f_1 = [1,5], f_2 = [6,10]\}$
$\phi_c = \{g_1 = [1,6], g_2 = [7,15]\}$

$\mathcal{P}_R = \{f_2\}\quad \mathcal{P}_S = \{g_1\}$

$\Delta R = \{\!\{\text{🔺}\ (5,8)\}\!\}$

Output for each incremental operator

Table access R	$\{\!\{\text{🔺}\ \langle(5,8),\{f_1\}\rangle\}\!\}$
Selection $\sigma_{a>3}$	$\{\!\{\text{🔺}\ \langle(5,8),\{f_1\}\rangle\}\!\}$
Join $\bowtie_{b=d}$	$\{\!\{\text{🔺}\ \langle(5,8,7,8),\{f_1,g_2\}\rangle\}\!\}$
Aggregation $\gamma_{\text{sum}(c);a}$	$\mathcal{S}[9] = (\text{SUM}=6, \text{CNT}=1, \mathcal{P}=\{f_2,g_1\}, \mathcal{F}_9=\{f_2:1, g_1:1\})$
	$\mathcal{S}'[9] = (\text{SUM}=6, \text{CNT}=1, \mathcal{P}=\{f_2,g_1\}, \mathcal{F}_9=\{f_2:1, g_1:1\})$
	$\mathcal{S}'[5] = (\text{SUM}=7, \text{CNT}=1, \mathcal{P}=\{f_1,g_2\}, \mathcal{F}_5=\{f_1:1, g_2:1\})$
	$\{\!\{\text{🔺}\ \langle(5,7),\{f_1,g_2\}\rangle\}\!\}$
Having $\sigma_{\text{sum}(c)>5}$	$\{\!\{\text{🔺}\ \langle(5,7),\{f_1,g_2\}\rangle\}\!\}$
Merging μ	$\mathcal{S}: \{f_2:1, g_1:1\}$
	$\mathcal{S}': \{f_1:1, f_2:1, g_1:1, g_2:1\}$
Sketch delta	$\text{🔺}\ \{f_1,g_2\}$

Figure 5: Using our IM to evaluate a query under incremental annotated semantics.

$$\mathcal{I}(Q_1 \times Q_2, \Delta\mathcal{D}) =$$
$$\{\!\{\text{🔺}\ \langle s \circ t, \mathcal{P}_1 \uplus \mathcal{P}_2\rangle^{n\cdot m} \mid (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)$$
$$\vee (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)$$
$$\vee (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \langle t,\mathcal{P}_2\rangle^m \in Q_2(\mathcal{D}))$$
$$\vee (\langle s,\mathcal{P}_1\rangle^n \in Q_1(\mathcal{D}) \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)\}\!\}$$
$$\cup$$
$$\{\!\{\text{🔺}\ \langle s \circ t, \mathcal{P}_1 \uplus \mathcal{P}_2\rangle^{n\cdot m} \mid (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)$$
$$\vee (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)$$
$$\vee (\text{🔺}\ \langle s,\mathcal{P}_1\rangle^n \in \Delta Q_1 \wedge \langle t,\mathcal{P}_2\rangle^m \in Q_2(\mathcal{D}))$$
$$\vee (\langle s,\mathcal{P}_1\rangle^n \in Q_1(\mathcal{D}) \wedge \text{🔺}\ \langle t,\mathcal{P}_2\rangle^m \in \Delta Q_2)\}\!\}$$

Continuing with Ex. 5.1, as $\Delta\mathcal{S} = \emptyset$ and $\Delta\mathcal{R} = \{\!\{\text{🔺}\ \langle(5,8),\{f_1\}\rangle\}\!\}$ only contains insertions, only $\Delta\mathcal{R} \bowtie_{b=d} \mathcal{S}$ returns a non-empty result (the third case above). As $(5,8)$ only joins with tuple $(7,8)$, a single delta tuple $\text{🔺}\ \langle(5,8,7,8),\{f_1,g_2\}\rangle$ is returned.

5.2.5 Aggregation: Sum, Count, and Average. For the aggregation operator, we need to maintain the current aggregation result for each individual group and record the contribution of fragments from a provenance sketch towards the aggregation result to be able to efficiently maintain the operator's result. Consider an aggregation operator $\gamma_{\mathbf{f}(a);G}(R)$ where $\mathbf{f}$ is an aggregation function and G are the group by attributes ($G = \emptyset$ for aggregation without group-by). Given a version R of the input of the aggregation operator, we use $\mathcal{G} = \{t.G | t \in R\}$ to denote the set of distinct group-by values.

The state data needed for aggregation depends on what aggregation function we have to maintain. However, for all aggregation functions the state maintained for aggregation is a map $\mathcal{S}$ from groups to a per-group state storing aggregation function results for this group, the sketch for the group, and a map $\mathcal{F}_g$ recording for each range ρ of Φ the number of input tuples belonging to the group with ρ in their provenance sketch. Intuitively, $\mathcal{F}_g$ is used in a similar fashion as for operator μ to determine when a range has to be added to or removed from a sketch for the group. We will discuss aggregation functions **sum**, **count**, and **avg** that share the same state.

Sum. Consider an aggregation $\gamma_{\text{sum}(a);G}(Q)$. To be able to incrementally maintain the aggregation result and provenance sketch for a group g, we store the following state:

$$\mathcal{S}[g] = (\text{SUM}, \text{CNT}, \mathcal{P}, \mathcal{F}_g)$$

SUM and CNT store the sum and count for the group, $\mathcal{P}$ stores the group's sketch, and $\mathcal{F}_g : \Phi \to \mathbb{N}$ introduced above tracks for each range $\rho \in \Phi$ how many input tuples from $Q(D)$ belonging to the group have ρ in their sketch. State $\mathcal{S}$ is initialized to $\emptyset$.

Incremental Maintenance. The operator processes an annotated delta as explained in the following. Consider an annotated delta $\Delta\mathcal{D}$. Let ΔQ denote $\mathcal{I}(Q, \Delta\mathcal{D})$, i.e., the delta produced by incremental evaluation for Q using $\Delta\mathcal{D}$. We use $\mathcal{G}_{\Delta Q}$ to denote the set of groups present in ΔQ and ΔQ_g to denote the subset of ΔQ including all annotated delta tuples $\Delta\langle t,\mathcal{P}\rangle$ where $t.G = g$. We now explain how to produce the output for one such group. The result of the incremental aggregation operators is then just the union of these results. We first discuss the case where the group already exists and still exists after applying the input delta.

Updating an existing group. Assume the current and updated state for g as shown below:

$$\mathcal{S}[g] = (\text{SUM}, \text{CNT}, \mathcal{P}, \mathcal{F}_g) \quad \mathcal{S}'[g] = (\text{SUM}', \text{CNT}', \mathcal{P}', \mathcal{F}'_g)$$

The updated sum is produced by adding $t.a \cdot n$ for each inserted input tuple with multiplicity n: $\text{🔺}\ \langle t,\mathcal{P}\rangle^n \in \Delta Q_g$ and subtracting this amount for each deleted tuple: $\text{🔻}\ \langle t,\mathcal{P}\rangle^n \in \Delta Q_g$. For instance, if the delta contains the insertion of 3 duplicates of a tuple with a value 5, then the SUM will be increased by $3 \cdot 5$.

$$\text{SUM}' = \text{SUM} \;+ \sum_{\text{🔺}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} t.a \cdot n \;- \sum_{\text{🔻}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} t.a \cdot n$$

The update for CNT is computed in the same fashion using n instead of $t.a \cdot n$. The updated count in $\mathcal{F}'_g$ is computed for each $\rho \in \Phi$ as:

$$\mathcal{F}'_g[\rho] = \mathcal{F}_g[\rho] + \sum_{\text{🔺}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge \rho \in \mathcal{P}} n \;- \sum_{\text{🔻}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge \rho \in \mathcal{P}} n$$

Based on $\mathcal{F}'_g$ we then determine the updated sketch for the group:

$$\mathcal{P}' = \{\rho \mid \mathcal{F}'_g[\rho] > 0\}$$

We then output a pair of annotated delta tuples that deletes the previous result for the group and inserts the updated result:

$$\text{🔻}\ \langle g \circ (\text{SUM}), \mathcal{P}\rangle \qquad \text{🔺}\ \langle g \circ (\text{SUM}'), \mathcal{P}'\rangle$$

Creating and Deleting Groups. For groups g that are not in $\mathcal{S}$, we initialize the state for g as shown below: $\mathcal{S}'[g] = (0, 0, \emptyset, \emptyset)$ and only output $\text{🔺}\ \langle g \circ (\text{SUM}'), \mathcal{P}'\rangle$. An existing group gets deleted if $\text{CNT} \neq 0$ and $\text{CNT}' = 0$. In this case we only output $\text{🔻}\ \langle g \circ (\text{SUM}), \mathcal{P}\rangle$.

Average and Count. For average we maintain the same state as for sum. The only difference is that the updated average is computed

as $\frac{\text{SUM}'}{\text{CNT}'}$. For count we only maintain the count and output CNT′. In [30], we also present the incremental annotated semantics for additional aggregation functions (**min** and **max**) which require maintaining a sort order over the tuples in each group to deal with deletion.

Continuing with Ex. 5.1, the output of the join (single delta tuple with group 5) is fed into the aggregation operator using **sum**. As no such group is in $\mathcal{S}$ we create new entry $\mathcal{S}[5]$. After maintaining the state, the output delta produced for this group is $\{\!\{\triangle\ \langle(5,7),\{f_1,g_2\}\rangle\}\!\}$. This result satisfies `HAVING` condition (selection $\sigma_{\mathbf{sum}(c)>5}$) and is passed on to the merge operator.

5.2.6 Top-k. The top-k operator $\tau_{k,O}$ returns the first k tuples sorted on O. As we are dealing with bag semantics, the top-k tuples may contain a tuple with multiplicity larger than 1. As before, we use ΔQ to denote $\mathcal{I}(Q,\Delta\mathcal{D})$.

State Data. To be able to efficiently determine updates to the top-k tuples with sketch annotations we maintain a nested map. The outer map $\mathcal{S}$ is ordered on O. It should be implemented using a data structure like balanced search trees (BSTs) that provide efficient access to entries in sort order. This map associates order-by values o with another map CNT which stores multiplicities for each annotated tuple $\langle t,\mathcal{P}\rangle$ for which $t.O = o$.

$$\mathcal{S}[o] = (\text{CNT})$$

and for any $\langle t,\mathcal{P}\rangle$ with $t.O = o$ with $\langle t,\mathcal{P}\rangle^n \in \Delta Q$ we store

$$\text{CNT}[\langle t,\mathcal{P}\rangle] = n$$

This data structure allows efficient updates to the multiplicity of any annotated tuple based on the input delta as shown below. Consider such a tuple $\langle t,\mathcal{P}\rangle$ with $t.O = o$ with $\triangle\ \langle t,\mathcal{P}\rangle^n \in \Delta Q$ and $\triangle\ \langle t,\mathcal{P}\rangle^m \in \Delta Q$.

$$\mathcal{S}'[o][\langle t,\mathcal{P}\rangle] = \mathcal{S}[o][\langle t,\mathcal{P}\rangle] + n - m$$

Computing Deltas. As k is typically relatively small, we select a simple approach for computing deltas by deleting the previous top-k and then inserting the updated top-k. Should the need arise to handle large k, we can use a balanced search tree and mark nodes in the tree as modified when updating the multiplicity of annotated tuples based on the input delta and use data structures which enable efficient positional access under updates, e.g., order-statistic trees [15]. Our simpler technique just fetches the first tuples in sort order from $\mathcal{S}$ and $\mathcal{S}'$ by accessing the keys stored in the outer map $\mathcal{S}$ in sort order. For each o we then iterate through the tuples in $\mathcal{S}[o]$ (in an arbitrary, but deterministic order since they are incomparable) keeping track of the total multiplicity m of tuples we have processed so far. As long as $m \leq k$ we output the current tuple and proceed to the next tuple (or order-by key once we have processed all tuples in $\mathcal{S}[o]$). Once $m \geq k$, we terminate. If the last tuple's multiplicity exceeds the threshold we output this tuple with the remaining multiplicity. Applied to $\mathcal{S}$ this approach produces the tuples to delete and applied to $\mathcal{S}'$ it produces the tuples to insert:

$$\triangle\ \tau_{k,O}(\mathcal{S}) \qquad\qquad \triangle\ \tau_{k,O}(\mathcal{S}')$$

5.3 Complexity Analysis

We now analyze the runtime complexity of operators. Let n denote the input delta tuple size and p denote the number of ranges of the partition on which the sketch is build on. For table access, selection, and projection, we need to iterate over these n annotated tuples to generate the output. As for these operations we do not modify the sketches of tuples, the complexity is $O(n)$. For aggregation, for each aggregation function we maintain a hashmap that tracks the current aggregation result for each group and a count for each fragment that occurs in a sketch for each tuple in the group. For each input delta tuple, we can update this information in $O(1)$ if we assume that the number of aggregation functions used in an aggregation operator is constant. Thus, the overall runtime for aggregation is $O(n \cdot p)$. For joins, we store bloom filters on the join attributes to pre-filter the input as we will discuss further in Sec. 6.2. Suppose the right input table has m tuples. Building such a filter incurs a one-time cost of $O(m)$ for scanning the table once. Consider the part where we join a delta of size n for the left input with the right table producing o output tuples. The cost this join depends on what join algorithm is used ranging from $O(n+m+o)$ for a hash join to $O(n \cdot m + o)$ for a nested loop join (in both cases assuming the worst case where no tuples are filtered using the bloom filter). For the top-k operator, we assume there are l nodes stored in the balanced search tree. Building this tree will cost $O(l \cdot \log l)$ (only built once). An insertion, deletion, or lookup will take $O(\log l)$ time. Thus, the runtime complexity of the top-k operator is $O(n \cdot \log l)$. Regarding space complexity, selection and projection only require constant space. For aggregation, the space is linear in the number of groups and in p. For join, the bloom filter's size is linear in m, but for a small constant factor. For top-k operators, we store $l \geq k$ entries in the search tree, each requiring $O(p)$ space. Thus, the overall space complexity for this operator is $O(l \cdot p)$.

5.4 Correctness Proof

We are now ready to state the main result of this paper, i.e., the incremental operator semantics we have defined is an incremental maintenance procedure. That is, it outputs valid sketch deltas.

THEOREM 5.1 (CORRECTNESS). *$\mathcal{I}$ as defined in Sec. 4.4 is an incremental maintenance procedure such that it takes as input a state $\mathcal{S}$, the annotated delta $\Delta\mathcal{D}$, the ranges Φ, a query Q and returns an updated state $\mathcal{S}'$ and a provenance sketch delta $\Delta\mathcal{P}$: $\mathcal{I}(Q,\Phi,\mathcal{S},\Delta\mathcal{D}) = (\Delta\mathcal{P},\mathcal{S}')$. For any query Q, sketch $\mathcal{P}$ that is valid for D, and state $\mathcal{S}$ corresponding to D we have:*

$$\mathcal{P}[Q,\Phi,D \uplus \Delta D] \subseteq \mathcal{P} \uplus \mathcal{I}(Q,\Phi,\mathcal{S},\Delta\mathcal{D})$$

PROOF SKETCH. We prove the statement by structural induction over the query demonstrating that applying μ to the result of the incremental interpretation of the query yields a valid sketch delta in the sense that applying this delta to the current version of the sketch yields an over-approximation of an accurate sketch for the updated database. The base case is a query consisting of single table access operator. For the inductive step, we assume that for all queries Q with up to n operators, $\mathcal{I}$ (ignoring μ applied at the end) (i) outputs the same set of tuples as Q under regular bag semantics and that (ii) each result tuple is annotated with a sketch that is sufficient for producing this tuple. Under this assumption we then show through a case distinction for each algebra operator that this invariant is preserved in the output of the operator evaluated on the annotated delta produced for Q. We present the full proof in [30]. □

6 The IMP System

While the semantics from Sec. 5 can be implemented in SQL, an in-memory implementation can be significantly more efficient as we can utilize data structures not available in a SQL-based implementation (see [30]). IMP is implemented as a stand-alone in-memory

engine that uses a backend database for fetching deltas and for evaluating operations (joins) that require access to large amounts of data. In Fig. 2, IMP's incremental engine is the pipeline shown in red. IMP executes $\mathcal{I}(Q, \mathcal{S}, \Delta\mathcal{D})$ to generate delta sketches. For joins (and cross products), $\Delta\mathcal{R} \bowtie \mathcal{S}$ and $\mathcal{R} \bowtie \Delta\mathcal{S}$ are executed by sending $\Delta\mathcal{R}$ ($\Delta\mathcal{S}$) to the database and evaluating the join in the database.

6.1 Storage Layout & State Data

We store data in a columnar representation for horizontal chunks of a table (*data chunks*). Annotated inserted / deleted tuples are stored in separate chunks. The annotations (provenance sketches) of the rows in a data chunk are stored in a separate column as bit sets.

Sketch & State Data. IMP stores sketches in a hash-table where the key is a query template for which the sketch was created and the value is the sketch and the state of the incremental operators for this query. Here a query template refers to a version of a query Q where constants in selection conditions are replaced with placeholders such that two queries that only differ in these constants have the same key. This is done to be able to efficiently prefilter candidate sketches to be used for a query as the techniques from [38] can determine whether a sketch for query Q_1 can be used to answer a query Q_2 if these queries share the same template. Furthermore, for each sketch we store a version identifier to record which database version the sketch corresponds to. IMP can persist its state in the database.

6.2 Optimizations

Data transfer between IMP and the DBMS can become a bottleneck. We now introduce several optimizations that reduce communication.

Bloom Filters For Join. For join operators, IMP uses the DBMS to compute the result of $\mathcal{R} \bowtie \Delta\mathcal{S}$ and $\Delta\mathcal{R} \bowtie \mathcal{S}$ which requires sending $\Delta\mathcal{R}$ (or $\Delta\mathcal{S}$) to the database. IMP maintains bloom filters on the join attributes for both sides of equi-joins that are used to filter out rows from $\Delta\mathcal{R}$ (and $\Delta\mathcal{S}$) that do not have any join partners in the other table. If according to the bloom filter no rows from the delta have join partners then we can avoid the round trip to the database.

Filtering Deltas Based On Selections. If a query involves a selection and all operators in the subtree rooted at a selection are stateless, then we can avoid fetching delta tuples from the database that do not fulfill the selection's condition as such tuples will neither affect the state of operators downstream from the selection nor will they impact the final maintenance result as their decedents will be filtered by the incremental selection. That is, we can push the selection conditions into the query that retrieves the delta.

Optimizing Minimum, Maximum, and Top-k. If the input to an aggregation with aggregation functions **min** or **max** or a top-k operator is large, then maintaining the sorted map used in the state of these operators to store all inputs can become a bottleneck. Instead of storing the full input, we can only store the top / bottom l tuples ($l > k$ for top-k and $l > 1$ for **min** and **max**). By keeping a record of the first l tuples, it is guaranteed that we can delete at least $l - k$ (or $l - 1$ for **min** and **max**) tuples from the input without having to recapture the sketch. In practice, as most tuples deleted will not the be in the top-k and updates contain both insertions and deletions, a moderately sized l is typically sufficient to completely avoid recapture.

6.3 Concurrency Control & Sketch Versions

So far we have assumed that the database backend uses snapshot isolation and that sketch versions are identified by snapshot identifiers. However, in snapshot isolation, each transaction sees data committed before it started (identified by a snapshot identifier) and its own changes. Thus, for a transaction that wants to use a sketch after updating a table accessed by the query for the sketch, we have to include the transaction's updates when maintaining the sketch. We can track these updates using standard audit logging mechanisms supported nativly in databases like Oracle or implemented through extensibility mechanisms like triggers to keep a history of row versions. For statement-level snapshot isolation (isolation level `READ COMMITTED` in systems like Postgres or Oracle), we face the challenge that even if we run the queries for incremental maintenance in the same transaction as the query that uses the updated sketch, these queries may see different versions of the database. Thus, supporting statement-level snapshot isolation requires either deeper integration into the database to run the maintenance query as of the same snapshot as the query that uses the sketch or the use of techniques like reenactment [5, 6] to reconstruct such database states.

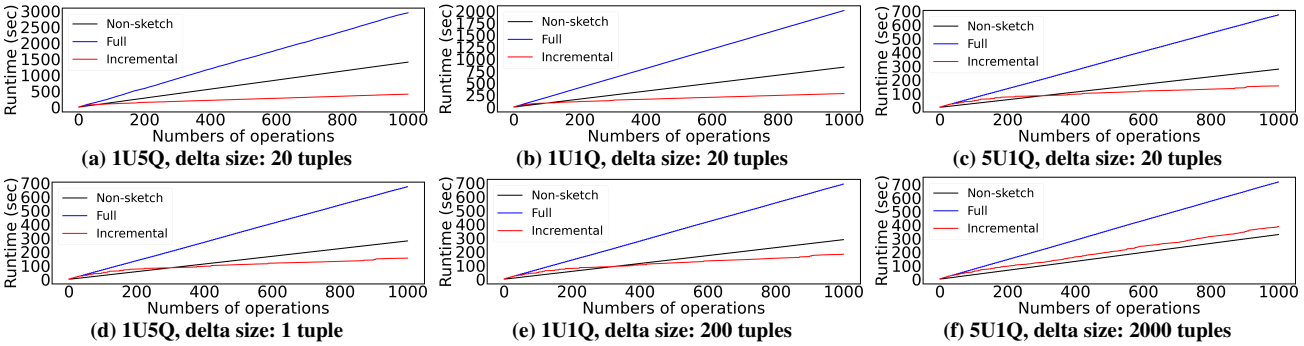
6.4 Selecting Sketch Attributes and Ranges

Both the choice of attribute for a sketch and the choice of ranges can affect the amount of data covered by a sketch. Regarding the choice of attribute, we first identify which attributes are safe using techniques from [38]. In [31] we studied cost-based selection of attributes for sketches. For IMP we employ a heuristic to select attributes based on the insights from [31]. We select attributes that are important for a query such as group-by attributes or attributes for which efficient access methods, e.g., an index, are available. Regarding the choice of ranges, as long as ranges are fine-granular enough and data is roughly evenly distributed across ranges, the exact choice of ranges has typically neglectable effect on sketch size. We use the bounds of equi-depth histograms maintained by many DBMS as statistics as ranges. Note that we generate ranges to cover the whole domain of an attribute instead of only its active domain. If a significant fraction of the data in a relation is updated, then this can lead to an imbalance in the amount of data per range and in turn to a degradation of the performance of sketches over time. As a significant change in distribution is unlikely to occur frequently, we can simply update the ranges and recapture sketches. In [30] we discuss potential strategies to avoid this.

7 Experiments

We evaluate IMP against two baselines: *full maintenance* and an approach that does not use PDBS to demonstrate the effectiveness of IMP to improve the performance of workloads that mix queries and updates. All experiments use a machine with 2 x 3.3Ghz AMD Opteron 4238 CPUs (12 cores) and 128GB RAM running Ubuntu 20.04 (linux kernel 5.4.0-96-generic) and Postgres 16.2. IMP's source code and the experimental setup are available at [29]. Experiments are repeated at least 10 times. We report median runtimes. The maximum variance was $< 5\%$ and $< 1\%$ in most experiments.

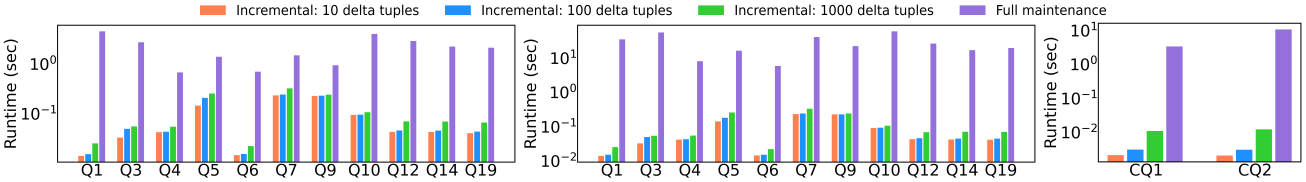
Datasets and Workloads. We use the *TPC-H* benchmark, a real world *Crime* dataset and a synthetic dataset (tables with 10M rows with at least 11 attributes). Each synthetic table has a key attribute



(a) 1U5Q, delta size: 20 tuples (b) 1U1Q, delta size: 20 tuples (c) 5U1Q, delta size: 20 tuples

(d) 1U5Q, delta size: 1 tuple (e) 1U1Q, delta size: 200 tuples (f) 5U1Q, delta size: 2000 tuples

Figure 6: Varying the delta size, we measure end-to-end workload runtime for NS, FM, and IMP.



(a) TPC-H 1GB (b) TPC-H 10GB (c) Crimes

Figure 7: Maintaining provenance sketches: incremental versus full maintenance on the TPC-H and Crime datasets.

`id`. For the other attributes, the values of one attribute (a) are chosen uniform at random. The remaining attributes are linearly correlated with a subject to Gaussian noise to create partially correlated values.

7.1 Mixed Workload Performance

In this experiment, we measure the end-to-end runtime of *IMP*, full maintenance (*FM*), and no-sketch (*NS*) on mixed workloads consisting of queries and updates. Both IMP and FM start without any sketches. The cost of maintaining and creating sketches is included in the runtime. Each workload consists of 1000 operations (each operation is either a query or an update). We refer to the ratio between queries and updates (the *query-update ratio*). We use the technique from [38] to determine whether an existing sketch for a query Q' can be used to answer the current query Q. If an existing sketch can be reused, we maintain the sketch if necessary. Otherwise, we create a new sketch. When an update on relation R is executed, we determine the delta and append it to the delta table for R, associating each tuple with a snapshot identifier. This enables us to fetch only delta tuples of updates that were executed after the sketch was last maintained.

We use a query template $Q_{endtoend}$ (see [30]) which is a group-by-aggregation-having query over the synthetic data and delta sizes 1, 20, 200 and 2000. We consider three query-update ratios: 1U1Q (one update per one query), 1U5Q (one update per five queries) and 5U1Q (five updates per one query). For full maintenance, whenever a sketch needs to be maintained, we recapture it. Fig. 6 shows the runtime for several combinations of query-update ratio and delta size (The x-axis indicates the total number of operations executed so far). We present additional combinations in [30]. FM has the highest cost: the cost of recapturing sketches frequently outweighs the benefit of using sketches. IMP outperforms both baselines, except for the extreme case 5U1Q with delta size 2000 (per update) where 5 updates affecting at least 10k tuples in total are executed between two adjacent queries. In the first part of each workload, the cost of creating provenance sketches outweighs the benefits of sketch use. However, once a sufficient set of sketches is available, PDBS outperforms the NS baseline.

Insights: IMP significantly improves the performance of mixed workloads using PDBS.

7.2 Incremental Versus Full maintenance

We now compare IMP against FM. We vary the *delta size* focusing on realistic delta sizes: 10, 50, 100, 500 and 1000.

7.2.1 TPC-H. The results for TPC-H (www.https://www.tpc.org/tpch/) at SF1 (~ 1 GB) and SF10 (~ 10GB) are shown in Fig. 7a and 7b. We selected queries that benefit from sketches [38] and are sufficiently complex (multiple joins, aggregation with `HAVING` or top-k). We turn on the selection push down and join bloom filter optimizations (see Sec. 6.2). The runtime of FM only depends on the current size of the database. Thus, we do not include results for different delta sizes for this method. IMP outperforms FM by at least a factor of 3.9 and up to a factor of ~2500, demonstrating the effectiveness of incremental maintenance. Importantly, the runtime of IMP, while depending on delta size, is mostly unaffected by database size as join is the only incremental operator accessing the database.

7.2.2 Crime Dataset. The Crime[4] dataset consists of a 1.87GB table with 7.3M incidents. We use two queries (see [30]): `CQ1`: The number of crimes per year and beat (geographical location). `CQ2`: Areas with more than 1000 crimes. As shown in Fig. 7c, IMP outperforms FM by at least 2 orders of magnitude (OOM).

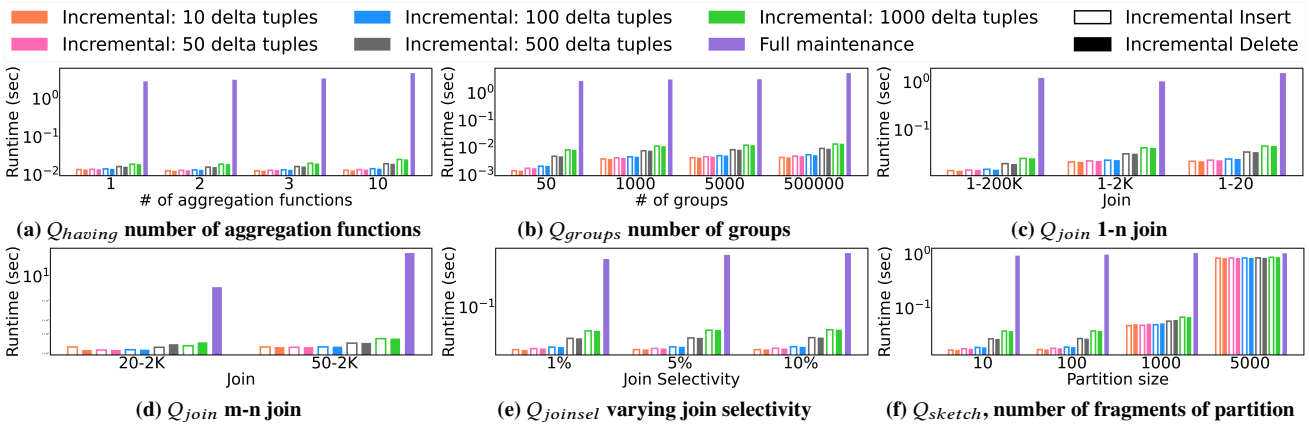
Insights: For deltas up to 1000 tuples, IMP outperforms FM by several OOM.

7.3 Microbenchmarks

Next, we evaluate in detail how IMP's performance is affected by various workload parameters using the synthetic dataset. We compare IMP against FM. The database size is kept constant, i.e., for FM, the runtime is not affected by varying the delta size.

7.3.1 Aggregation Functions and Groups. We use query template Q_{groups} (see [30]) that is a group-by aggregation query with `HAVING` over a single table and vary the number of groups:

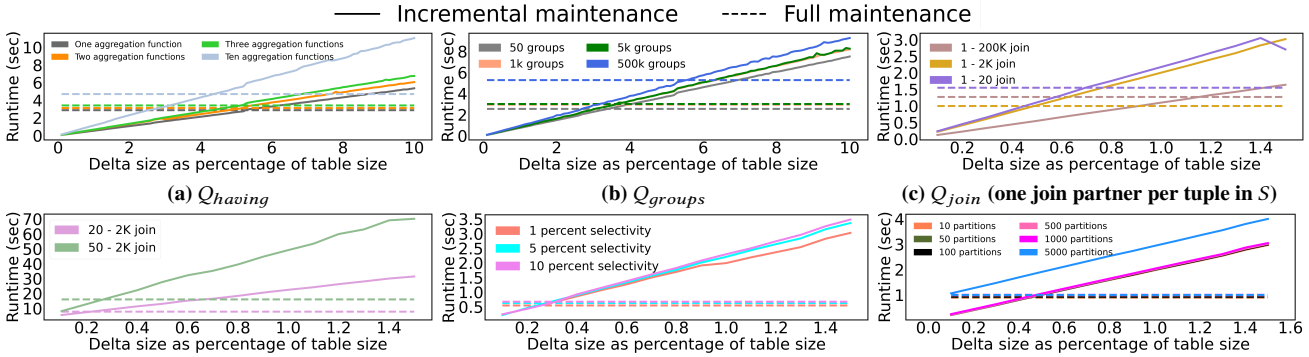
[4]https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2



(a) Q_{having} number of aggregation functions (b) Q_{groups} number of groups (c) Q_{join} 1-n join

(d) Q_{join} m-n join (e) $Q_{joinsel}$ varying join selectivity (f) Q_{sketch}, number of fragments of partition

Figure 8: Microbenchmarks ("realistic" delta size): varying delta size from 10 tuples to 1000 tuples.



(a) Q_{having} (b) Q_{groups} (c) Q_{join} **(one join partner per tuple in** S**)**

(d) Q_{join} **(varying #join partners per tuple in** S**)** (e) $Q_{joinsel}$ (f) Q_{sketch}

Figure 9: Microbenchmarks: varying delta size to determine the "break even point" where FM outperforms IMP.

50, $1K$, $5K$ and $500K$. As the state $\mathcal{S}$ for aggregation contains an entry for each group, we expect that runtime will increase when increasing the number of groups. As shown in Fig. 8b, for delta sizes up to 1000 tuples, IMP outperforms FM by 100x (500k groups) to 1000x (50 groups). Fig. 9b shows that the break even point lies at delta sizes between ~3.5% (for 50 group) and ~ 5.5% for (500k groups). While the runtime of IMP increases when increasing the number of groups, the effect is more pronounced for FM that computes results for all groups. Fig. 8a and Fig. 9a show the runtime of IMP vs. FM for a group-by having query with 5k groups, varying the number of aggregation functions. IMP outperforms FM by up to ~ 100x for delta sizes up to ~ 5% of the database.

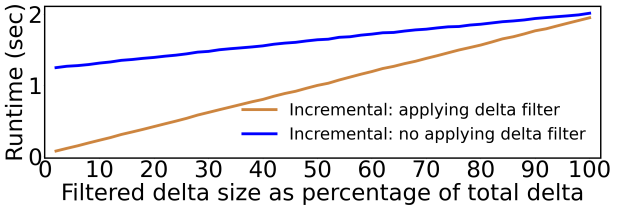
7.3.2 Joins. We evaluate group-by aggregation queries with `HAVING` over the result of an equi-join using query template Q_{join} (see [30]). Both input tables have $10M$ rows. The synthetic tables are designed as the follows: for an $m-n$ join $R \bowtie S$, the selectivity is 100% for table S, and there are $10^8/n$ distinct join attribute values with a multiplicity of n; for the other table R, there are m tuples that join with each distinct join attribute value in S. For instance, the result size for $2-2k$ as well as for $2-200k$ is $2 \cdot 10M = 20M$ tuples.

Fig. 8c and Fig. 9c show the runtime of incremental vs. FM for $1-n$ joins, and Fig. 8d and Fig. 9d show results for $m-2K$ joins. In the 1-n join experiment, the 1-20 join is more expensive than the 1-20k and 1-200k joins because even through the 1-20 join has less join result tuples, there are more groups for the aggregation functions above the join operator as the join attribute of table S is also the group-by attribute for the query. There are $10^8/20$ distinct groups and each group has a multiplicity of 20. For the m-n join, the queries all have the same number of groups. 50-2k has more join results to process and, thus, is slower than 20-2k join.

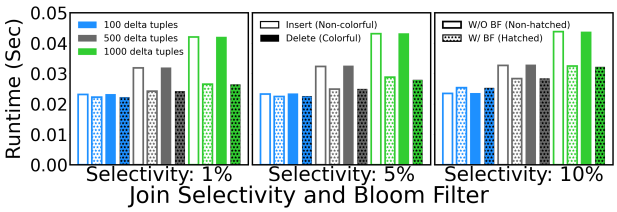
Recall from Sec. 7.2.1, that IMP computes $\Delta R \bowtie S$ by running a SQL query. Thus, incrementally maintaining joins requires sending all delta tuples for the join inputs to the DBMS. That is, the break even point is lower for Q_{join} than for Q_{having}. Our bloom-filter optimization for joins can sometimes avoid an additional round trip to the database for those tuples that do not have the join partner. We further evaluate this optimization in Sec. 7.4.

To evaluate performance of queries with more selective joins, we use a group-by-aggregation query over a join: $Q_{joinsel}$ (R join S, see [30]) and vary join selectivity: 1%, 5%, and 10%. Fig. 8e and Fig. 9e show the runtime of IMP and FM. For small deltas, the join selectivity has a smaller impact on IMP than for larger deltas as for small deltas we are joining a small table ($\Delta\mathcal{R}$) with a large table ($\mathcal{S}$), i.e., the bottleneck is scanning the large table.

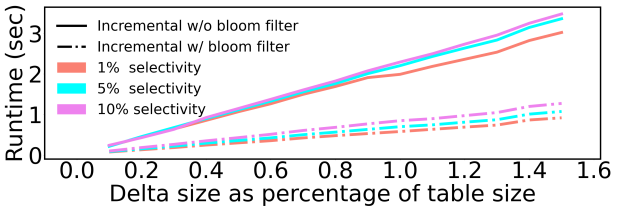
7.3.3 Varying Partition Granularity. We now vary #frag, the number of fragments in the sketch's partition. We use template Q_{sketch} (see [30]) which is a group-by aggregation query with `HAVING` over the results of a join. We vary the number of fragments #frag from 10 to 5000. Fig. 8f and Fig. 9f show the runtime for IMP and FM. While the cost of FM is impacted by #frag, the dominating cost is evaluating the full capture query, resulting in an insignificant runtime increase when #frag is increased. In contrast, incremental maintenance cost increases linearly in the delta size.



(a) Optimization: filter delta



(b) Optimization: bloom filter realistic delta size



(c) Optimization: bloom filter large delta size

Figure 10: Optimizations in IMP: filtering deltas based on selection conditions and using bloom filters for joins.

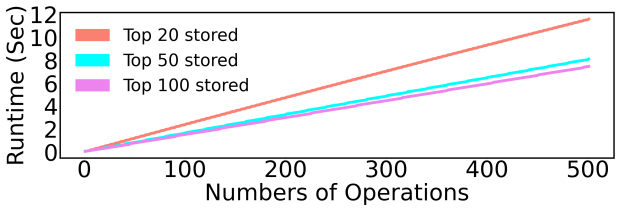
7.4 Optimizations

7.4.1 Selection push-down for deltas. We evaluate the effectiveness of our delta selection push-down optimization that filters the delta based on selection conditions in the query. We use query Q_{selpd} which is a group-by aggregation query (see [30]) and vary the selectivity of the query's `WHERE` clause. We fix delta size to 2.5% of the table and vary the fraction of delta tuples that fulfills the condition from 2% to 100%. The results (Fig. 10a) demonstrate that the cost of filtering delta tuples is amortized by reducing maintenance cost and communication with the DBMS.

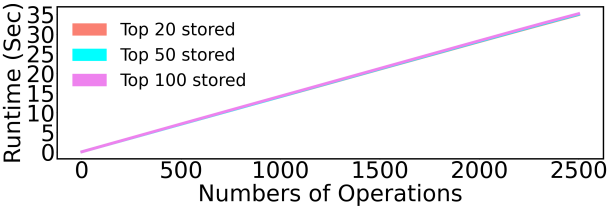
7.4.2 Join optimization using bloom filters. Another optimization we applied in IMP is to use bloom filters to track which tuples potentially have join partners. As in the microbenchmarks for join, we use query $Q_{joinsel}$ (see [30]). As shown in Fig. 10b and 10c, filtering the delta using bloom filters is effective for all delta sizes, even for larger selectivity, due to (i) the reduction in data transfer between IMP and the database, (ii) the reduction of the input size for the query evaluating $\Delta\mathcal{R} \bowtie \mathcal{S}$ by reducing the size of $\Delta\mathcal{R}$, and (iii) reducing the input size for incremental operators.

Insights: IMP's performance is mainly impacted by delta size. As join requires a round trip to the database, queries with join are typically more expensive. Our bloom filter optimization reduces this cost. Nonetheless, IMP significantly outperforms FM.

7.4.3 Top-K operator. For a top-k operator (Sec. 5.2.6) we store its inputs in an ordered map to be able to deal with deletions that remove a tuple from the current top-k. In practice keeping a buffer of the top-l tuples for $l > k$ is often sufficient. The potential drawback is that if all tuples from the buffer are deleted, we have to recapture the sketch. To evaluate this trade-off we run an experiment varying l (20, 50 and 100). The query we use is a top-10 query Q_{top-k} (see [30]). We then evaluate workloads



(a) Deleting from top-k groups



(b) Deleting random tuples

Figure 11: Top-k runtime, varying buffer size l

that delete data (20 tuples per update) from the table (the table contains 50k tuples and 5k distinct group-by values). We consider two extremes: (i) always delete tuples contributing to the top-k and (ii) randomly delete tuples. If less than k groups remain in the state data structure, IMP has to recapture the sketch. Fig. 11 shows the runtime varying l. For the worst case workload (only deleting tuples from the top-k), the additional cost of maintaining a larger state for the top-k operator is amortized by reducing the frequency of recapture. For the other extreme (uniform deletion), recapture is rarely needed. Overall, larger buffer sizes l can be recommended [30].

Maintenance Strategies. In [30] we also evaluate the impact of batch size on the cost of eager maintenance, demonstrating that batch sizes below 50 tuples should be avoided. In general, lazy maintenance is superior as we delay maintenance as long as possible and avoid maintenance of sketches that are not used.

8 Conclusions And Future Work

We present the first approach for in-memory incremental maintenance of provenance sketches. Our IMP system implements incremental maintenance rules for sketch-annotated data. Our experimental results demonstrate the effectiveness of our approach and optimizations, outperforming full maintenance by several orders of magnitude. In future work, in addition to extending IMP with support for more operators, e.g., set difference, recursive queries, or kNN search in vector databases, we will investigate how to integrate provenance-based data skipping and incremental maintenance of sketches with cost-based query optimization and self-tuning. Another open research question is how IMP can be extended as a general IVM engine for provenance information. Furthermore, IMP can be extended for maintaining summarizes of provenance [3, 4, 28] which, like sketches, can tolerate approximation and are typically small. One application would be data integration where we want to track the set of data sources a query result depends on.

Acknowledgments

This work was supported by NSF Awards IIS #2420577 and IIS #2420691.

Artifacts

IMP's source code (folder `IMP engine`), experimental scripts, and data (folder `dataset`) are provided in [29].

References

[1] Martín Abadi, Frank McSherry, and Gordon D. Plotkin. 2015. Foundations of Differential Dataflow. In *ETAPS*, Vol. 9034. 71–83.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505.

[3] Eleanor Ainy, Pierre Bourhis, Susan B. Davidson, Daniel Deutch, and Tova Milo. 2015. Approximated Summarization of Data Provenance. In *CIKM*. 483–492.

[4] Omar AlOmeir, Eugenie Yujing Lai, Mostafa Milani, and Rachel Pottinger. 2021. Summarizing Provenance of Aggregate Query Results in Relational Databases. In *ICDE*. 1955–1960.

[5] Bahareh Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2016. Reenactment for Read-Committed Snapshot Isolation. In *CIKM*. 841–850.

[6] Bahareh Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2018. Using Reenactment to Retroactively Capture Provenance for Transactions. *TKDE* 30, 3 (2018), 599–612.

[7] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bull.* 41, 1 (2018), 51–62.

[8] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. 2005. An annotation management system for relational databases. *VLDBJ* 14, 4 (2005), 373–396.

[9] José A. Blakeley, Per Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *SIGMOD*. 61–71.

[10] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *PVLDB* 16, 7 (2023), 1601–1614.

[11] Peter Buneman and Eric K. Clemons. 1979. Efficiently Monitoring Relational Databases. *TODS* 4, 3 (1979), 368–382.

[12] Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance. In *VLDB*. 577–589.

[13] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *ICDE*. 190–200.

[14] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for Deferred View Maintenance. In *SIGMOD*. 469–480.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

[16] Stefan Fehrenbach and James Cheney. 2018. Language-integrated provenance. *Sci. Comput. Program.* 155 (2018), 103–145.

[17] Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. 1992. Implementation of Delayed Updates in Heraclitus. In *EDBT*, Vol. 580. 261–276.

[18] Boris Glavic. 2021. Data Provenance - Origins, Applications, Algorithms, and Models. *Foundations and Trends® in Databases* 9, 3-4 (2021), 209–441.

[19] Boris Glavic, Renée J. Miller, and Gustavo Alonso. 2013. Using SQL for Efficient Generation and Querying of Provenance Information. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, Vol. 8000. 291–320.

[20] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *SIGMOD*. 328–339.

[21] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. 1992. Counting solutions to the View Maintenance Problem. In *Workshop on Deductive Databases*, Vol. CITRI/TR-92-65. 185–194.

[22] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[23] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *SIGMOD*. 157–166.

[24] Xiao Hu and Stavros Sintos. 2024. Finding Smallest Witnesses for Conjunctive Queries. In *ICDT*. 24:1–24:20.

[25] Grigoris Karvounarakis and Todd J. Green. 2012. Semiring-annotated data: queries and provenance? *SIGMOD Rec.* 41, 3 (2012), 5–14.

[26] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDBJ* 23, 2 (2014), 253–278.

[27] Volker Küchenhoff. 1991. On the Efficient Computation of the Difference Between Concecutive Database States. In *DOOD*, Vol. 566. 478–502.

[28] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2020. Approximate Summaries for Why and Why-not Provenance. *PVLDB* 13, 6 (2020), 912–924.

[29] Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic, and Xing Niu. 2025. Experiments and Source Code Repository. https://github.com/IITDBGroup/IMP_EDBT26.

[30] Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic, and Xing Niu. 2025. In-memory Incremental Maintenance of Provenance Sketches (extended version). arXiv:2505.20683.

[31] Ziyu Liu and Boris Glavic. 2025. Cost-based Selection of Provenance Sketches for Data Skipping. *CoRR* abs/2504.19252 (2025). arXiv:2504.19252

[32] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD*. 307–318.

[33] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[34] Haneen Mohammed, Charlie Summers, Sughosh Kaushik, and Eugene Wu. 2023. SmokedDuck Demonstration: SQLStepper. In *SIGMOD*. 183–186.

[35] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In *AAAI*. 1560–1568.

[36] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. 439–455.

[37] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. 2016. Incremental, Iterative Data Processing With Timely Dataflow. *Commun. ACM* 59, 10 (2016), 75–83.

[38] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. 2021. Provenance-based Data Skipping. *PVLDB* 15, 3 (2021), 451–464.

[39] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. 2019. Heuristic and Cost-Based Optimization for Diverse Provenance Tasks. *TKDE* 31, 7 (2019), 1267–1280.

[40] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, and Venkatesh Radhakrishnan. 2017. Provenance-Aware Query Optimization. In *ICDE*. 473–484.

[41] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*. 802–813.

[42] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *CoRR* abs/1801.07237 (2018). arXiv:1801.07237

[43] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *PVLDB* 11, 12 (2018), 2034–2037.

[44] Oded Shmueli and Alon Itai. 1984. Maintenance of Views. In *SIGMOD*. 240–255.

[45] Dimitra Vista. 1994. View maintenance in relational and deductive databases by incremental query evaluation. In *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*. 70.

[46] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDBJ* 12, 3 (2003), 262–283.

[47] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*. 615–626.

[48] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. 2004. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *ICAC*. 180–188.