In-memory Incremental Maintenance of Provenance Sketches

Pengyuan Li*, Boris Glavic$^{\alpha}$, Dieter Gawlick$^{\beta}$, Vasudha Krishnaswamy$^{\beta}$, Zhen Hua Liu$^{\beta}$, Danica Porobic$^{\beta}$, Xing Niu$^{\beta}$
Illinois Institute of Technology*, University of Illinois Chicago$^{\alpha}$,Oracle$^{\beta}$
pli26@hawk.iit.edu,bglavic@uic.edu
{dieter.gawlick,vasudha.krishnaswamy,zhen.liu,danica.porobic,xing.niu}@oracle.com

ABSTRACT

Provenance-based data skipping [30] compactly over-approximates the provenance of a query using so-called provenance sketches and then utilizes this information to speed-up the execution of subsequent queries by skipping irrelevant data. However, a provenance sketch captured at some time in the past may no longer correctly reflect what data is relevant for a query if the data has been updated subsequently. Thus, there is a need to maintain provenance sketches under updates. In this work, we introduce **I**n-Memory incremental **M**aintenance of **P**rovenance sketches (IMP), a framework for maintaining sketches incrementally when the data is updated. At the core of IMP is an incremental query engine for data annotated with provenance sketches that exploits the coarse-grained nature of sketches to enable novel optimizations that trade performance of maintenance for sketch size which in turn affects the performance of query answering with sketches. We demonstrate experimentally that incremental maintenance significantly reduces the cost of sketch maintenance for a wide range of workloads.

PVLDB Reference Format:
Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic, Xing Niu. In-memory Incremental Maintenance of Provenance Sketches. PVLDB, (): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IITDBGroup/2025_PVLDB_IMP.

1 INTRODUCTION

Database engines take advantage of physical design such as index structures, zone maps [25] 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 [30], 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 [30] we introduced such a dynamic relevance analysis technique called *provenance-based data skipping* (PBDS). In PBDS, 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. We have presented safety conditions in [30] that ensure that 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. Generating a provenance sketch for a query Q (which we refer to as *capturing* the sketch) requires evaluating an instrumented version of Q that returns the desired sketch. This additional cost is amortized over time by utilizing the sketch to answer subsequent queries by filtering data early-on that does not belong to the sketch. Importantly, utilizing the techniques from [30], a sketch created for a query Q_1 can often be used to answer a different query Q_2 that has the same structure (differs only in constants used in conditions, e.g., multiple executions of the same parameterized query). Similar to query answering with materialized views, PBDS has to pay an upfront cost (creating the sketch) to provide future gains by reducing the execution time of queries that can utilize an existing sketch. However, as shown in [30], the trade-offs for provenance sketches differ from the trade-offs for materialized views as provenance sketches can reuse existing physical design and require almost no extra storage (typically less than 1KB per sketch).

As demonstrated in [30], PBDS enables database systems to exploit physical design for new classes of queries. However, just like a materialized view, a provenance sketch captured at some point of time in the past may no longer correctly reflect what data is needed (has become *stale*) when the database is updated. Either the sketch has to be maintained to be valid for the current database state or we have to drop the sketch.

EXAMPLE 1.1. *Consider the sales database shown Fig. 1 and query Q_{Top} that returns products whose total sale volume is greater than \$5000. Evaluating this query over the table shown in Fig. 1 produces a single result tuple* $(Apple, 5074)$. *The provenance of Q_{Top} over table* `sales` *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 have to select a range-partition of the* `sales` *table (the range partition does not necessarily have to correspond to the physical storage layout of the table). For instance, we may choose F_{price} that*

Proceedings of the VLDB Endowment, Vol. , No. ISSN 2150-8097.
doi:XX.XX/XXX.XX

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	5	f_2
s_7	7	HP	HP ProBook 550 G9	899	1	f_2

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

partitions the table on attribute `price` *based on ranges:*

$$\mathcal{R}_{price} = \{r_1 = [1, 600], r_2 = [601, 1000], r_3 = [1000, 1501], r_4 = [1501, 10000]\}$$

In Fig. 1, we show the fragment f_i for the range r_i each tuple belongs to on the right of the tuple. Two fragments (f_3 and f_4 highlighted in red) contain provenance and, thus, the provenance sketches for Q_{Top} wrt. $F_{\textbf{sales},price}$ is $\mathcal{P} = \{r_3, r_4\}$. Evaluating the query over the data described by the sketch is guaranteed to result in the same result as evaluating the query over the full database.[1] *Creating (capturing) a sketch requires execution of an instrumented version of a query. This cost is amortized over time by using the sketch to answer future queries such as repeated executions of Q_{Top} or queries with the same structure, i.e., using a different threshold in the* `HAVING` *clause.*

As demonstrated in [30], provenance-based data skipping can significantly improve query performance — we pay 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 based on the sketch. For instance, consider the sketch for Q_{Top} from Ex. 1.1 containing two ranges $r_3 = [1001, 1500]$ and $r_4 = [1501, 10000]$. To filter data not belonging to the sketch, we create a disjunction of conditions testing that each tuple passing the `WHERE` clause has a price within r_3 or r_4:[2]

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

A sketch may become stale when the database is updated after the sketch has been created. 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 incremental maintenance techniques for sketches. Sketches are captured using annotation propagation techniques to instrument a query. In a first step, for each input tuple the instrumented query determines which fragment the tuple belongs to. The singleton set containing this fragment is the initial sketch for the tuple. These sketches are then propagated and merged such that the final result of the instrumented query is the query's sketch.

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 r_2 as fragment f_2 contains these tuples. Evaluating Q_{Top} over the outdated sketch leads to an incorrect result that misses the group for HP. It is necessary to maintain the sketch, to ensure that it can be used to compute correct query answers over the current database.*

A straightforward approach to maintain provenance sketches under updates is *full maintenance* which means that we will drop all existing sketches and capture them again from scratch by evaluating each sketch's *capture query* (the instrumented query that computes the sketch) over the new version of the database. Typically, capture queries are more expensive than the queries for which sketches are generated. Thus, frequent execution of capture queries is not feasible. Note that capture queries can be expressed in SQL. Consider a partition F of a table R accessed by Q and let $Q_{R,F}$ denote this query, generated using the rewrite rules from [30]. Existing view maintenance techniques can be used to incrementally maintain the result of $Q_{R,F}$, the sketch generated by this query. However, this disregards the nature of this query which propagates partial sketches alongside intermediate query results as *annotations* on rows. 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 the query over the full database. It is often possible to further over-approximate the sketch, trading improved maintenance performance for increased sketch size.

We start by formalizing 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 the capture query for a user query Q to update the sketch created by the capture query 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 and (ii) filtering deltas for joins using bloom filters.

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

[1] In general, this is not the case for non-monotone queries. The safety check from [30] can be used to test whether a particular partition for a table is guaranteed to yield a safe sketch for a query Q and the partition used here passes this safety check for Q_{Top}.

[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`.

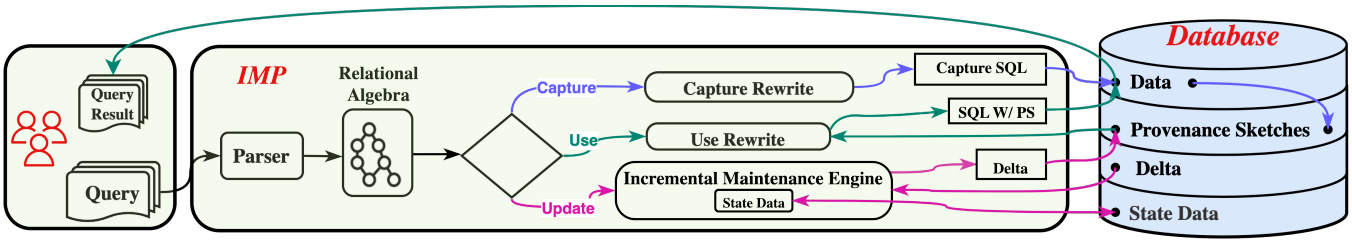


Figure 2: IMP manages a set of sketches. For each incoming query, IMP determines whether to (i) capture a new sketches, (ii) use an existing non-stale sketch, or (iii) incrementally maintain a stale sketch and then utilize the updated sketch to answer the query.

- We develop incremental versions of relational algebra operators for sketch-annotated data, and come up with rules for incrementally maintain sketches for operators.
- We implement these operator rules in IMP, an in-memory engine for incremental sketch maintenance, and present several optimizations to improve the performance of IMP.
- We experimentally compare IMP against an SQL-based approach of full maintenance and against a baseline that does not use provenance sketches. Our experimental evaluation over TPC-H dataset, real world datasets and synthetic data demonstrates the effectiveness of our engine and of our optimizations. IMP significantly outperforms full maintenance, often by several orders of magnitude.

The remainder of this paper is organized as follows: an overview of IMP is presented in Sec. 2. We discuss related work in Sec. 3. Sec. 4 covers relevant background and formally defines the incremental maintenance problem for sketches as well as our annotated data model. In Sec. 5, we introduce incremental sketch maintenance rules for relational operators and demonstrate the correctness of these rules. We discuss our implementation of IMP in Sec. 7 and present experimental results in Sec. 8. We conclude in Sec. 9.

2 OVERVIEW OF IMP

Fig. 2 shows a overview of IMP that operates as a middleware between the user and a DBMS. 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 queries, and the database version it was last maintained at or first captured at for sketches that have not been maintained yet. We assume that the DBMS uses snapshot isolation and we can use snapshot identifiers used by the database internally. 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 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. 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.

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 [30] 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, then we determine whether $\mathcal{P}$ is stale which can happen if the lazy maintenance strategy is applied. If that is the case, then IMP incrementally maintains the sketch (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 (green pipeline). 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 $\mathcal{P}$ (blue pipeline). This sketch is then used to answer Q.

3 RELATED WORK

Provenance. Provenance can be captured by annotating data and propagating these annotations using relational queries or by extending the database system [20] [32] [31]. Systems like GProM [5], Perm [15], Smoke [34], Smoked Duck [26], Links [13], ProvSQL[35] and DBNotes[6] capture provenance for SQL queries. In [30], 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 [7, 9, 14, 19, 22, 36]. [18, 37] gives an overview of many techniques and applications of view maintenance. Early work on view maintenance, e.g., [7, 9], used set semantics. This was later expanded to bag semantics (e.g., [11, 16]). We consider bag semantics. Materialization has been studied for Datalog

as well [17, 19, 27]. Incremental maintenance algorithms for iterative computations have been studied in [1, 8, 28, 29]. [21] proposed higher-order IVM. [38] maintains aggregate views in temporal databases. [33] proposes a general mechanism for aggregation functions. [2, 40] studied automated tuning of materialized views and indexes in databases. Our work has in common with self-tuning of physical design that we have to decide when it is worth to pay the overhead of creating a physical design artifact (a provenance sketch in our case) based on predication of future benefits resulting from using the artifact for speeding up operations. Several strategies have been studied for maintaining views eagerly and lazily. For instance, [12] presented algorithms for deferred maintenance and [7, 10, 19] 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. [39] presents a system for maintenance of provenance in a distributed Datalog engine. In contrast to our work, [39] is concerned with efficient distributed computation and storage or provenance. We focus on incremental maintenance of provenance sketches which, because of their small size and coarse-grained nature, enables new optimizations. Furthermore, as our goal of using provenance is improving query performance, we need to ensure that incremental maintenance is cheap enough to be amortized by utilizing the maintained sketches.

4 NOTATION AND BACKGROUND

In this section we introduce necessary background on relational algebra, and provenance sketch, and describe the notations in this paper. A database schema is a set of relation schemas. A relation schema of arity n consists of a name and a list of attribute names a_1 to a_n. Let $\mathbb{U}$ be a domain of values. An instance R of an n-ary relation schema $\text{SCH}(R)$ 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 tuple t with multiplicity n in relation R, i.e., $R(t) = n$. Fig. 3 shows the bag semantics relational algebra used in this work. We use $\text{SCH}(Q)$ to denote the schema of the result of query Q and $Q(I)$ to denote the result of evaluating query Q over database instance I. 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. $R \times S$ 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). Top-K $\tau_{k,o}(R)$ 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: $\mathbf{pos}(t, R, O) = \sum_{t' <_O t \wedge t^m \in R} m$.

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

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

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

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

Figure 3: 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 $A \subset \mathbf{R}$, range partitioning determines membership of tuples to fragments based on which interval their values belong to. For simplicity, we define range partitioning for a single attribute a.

DEFINITION 4.1 (RANGE PARTITION). *Consider a relation R and $a \in \mathbf{R}$. Let $\mathbb{D}(a)$ denote the domain of a. Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of intervals $[l, u] \subseteq \mathbb{D}(a)$ such that $\bigcup_{i=0}^{n} r_i = \mathbb{D}(a)$ and $r_i \cap r_j = \emptyset$ for $i \neq j$. The* range-partition *of R on a according to $\mathcal{R}$ denoted as $F_{\mathcal{R},a}(R)$ is defined as:*

$$F_{\mathcal{R},a}(R) = \{R_{r_1}, \ldots, R_{r_n}\} \quad \textbf{\textit{where}} \quad R_r = \{\!|t^n \mid t^n \in R \wedge t.a \in r|\!\}$$

In the following we will write F instead of $F_{\mathcal{R},a}$ if $\mathcal{R}$ and a are clear from the context or irrelevant to the discussion. Furthermore, we will use f to denote a fragment, e.g., we may write $F = \{f_1, \ldots, f_n\}$. We also extend range partitions to databases. For a database $D = \{R_1, \ldots, R_n\}$, we use $\mathcal{D}$ to denote a set of range - attribute pair $\{(\mathcal{R}_1, a_1), \ldots, (\mathcal{R}_n, a_n)\}$ such that $F_{\mathcal{R}_i,a_i}$ is a partition for R_i. Note that we do not require that all relations of D are associated with a sketch. This is modeled by setting $\mathcal{R}_i = \{[min(\mathbb{D}(a_i)), max(\mathbb{D}(a_i))]\}$, a single range covering all domain values, for relations that are not part of a sketch.

4.1.2 Provenance Sketches. Consider a database D, query Q, and a range partition $F_{\mathcal{D}}$ of D. A provenance sketch $\mathcal{P}$ for Q according to $\mathcal{D}$ is a subset of the ranges $\mathcal{R}_i$ for each $\mathcal{R}_i \in \mathcal{D}$ such that the fragments corresponding to the ranges in $\mathcal{P}$ fully cover Q's provenance within each R_i in D, i.e., $P(Q, D) \cap R_i$. Abusing notation, we will pretend that $\mathcal{D}$ is a single set of ranges, e.g., we may write $r \in \mathcal{D}$ instead of $r \in \mathcal{R}_i$ for $\mathcal{R}_i \in \mathcal{D}$ and D_r for r from $\mathcal{R}_i$ to denote the subsets of the database where all relations are empty except for R_i which is set to $R_{i,r}$, the fragment for r. We use $\mathcal{R}(D, \mathcal{D}, Q) \subseteq \mathcal{D}$ to denote the set of ranges whose fragments contains at least one tuple from the provenance denoted as $P(Q, D)$:

$$\mathcal{R}(D, \mathcal{D}, Q) = \{r \mid r \in \mathcal{R}_i \wedge \exists t \in P(Q, D) : t \in R_{i,r}\}$$

DEFINITION 4.2 (PROVENANCE SKETCH). *Let Q be a query, D a database, R a relation accessed by Q, and $\mathcal{D}$ a range partition of D. We call a subset $\mathcal{P}$ of $\mathcal{D}$ a* ***provenance sketch*** *iff $\mathcal{P} \supseteq \mathcal{R}(D, \mathcal{D}, Q)$. A sketch is called* ***accurate*** *if $\mathcal{P} = \mathcal{R}(D, \mathcal{D}, Q)$. We use $D_{\mathcal{P}}$, called the* ***instance*** *of $\mathcal{P}$, to denote $\bigcup_{r \in \mathcal{P}} D_r$.*

Consider the database consisting of a single relation (`sales`) from our running example shown in Fig. 1. According to the partition

$\mathcal{D} = \{(\mathcal{R}_{price}, price)\}$, the accurate provenance sketch $\mathcal{P}$ for the query Q_{Top} according to $\mathcal{D}$ consists of the set of ranges $\{r_3, r_4\}$ (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: $\{s_3, s_4, s_5\}$.

4.2 Updates, Histories, and Deltas

We model an update operation U as a function that takes as input a database D and return an updated database $U(D)$. This is sufficient for modeling SQL updates, insert, and delete statements. A history $\mathcal{W}$ is a sequence of updates. We use D_i to denote the version of the database after executing U_i starting from $D_0 = \emptyset$:

$$\mathcal{W} = (U_1, \ldots, U_n) \qquad D_0 = \emptyset \qquad D_i = U_i(D_{i-1})$$

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 $\{\!\{\mathbb{D}t \mid \mathbb{D}t \in \Delta D\}\!\}$ ($\{\!\{\mathbb{A}t \mid \mathbb{A}t \in \Delta D\}\!\}$). Given a database history $\mathbf{D}$, we will use ΔD_i to denote $\Delta(D_i, D_{i-1})$ and ΔD_{ij} to denote $\Delta(D_i, D_j)$. A delta ΔD can be *applied* to a database D denoted as $D \uplus \Delta D$:

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

4.3 Provenance Sketches Capture and Deltas

Given a database D and a query Q and partitioning ranges $\mathcal{D}$ we use $\mathcal{P}[Q, \mathcal{D}, D]$ to denote an **accurate provenance sketch** $\mathcal{P}$ for Q wrt. to D, and ranges $\mathcal{D}$. Such sketches can be created using the techniques from [30]. For sketches we use the same delta notation as for databases, e.g., $\mathcal{P}_i$ denotes the version of a sketch wrt. database version D_i, and $\Delta\mathcal{P}_{ij}$ denotes $\Delta(\mathcal{P}_i, \mathcal{P}_j)$. Furthermore, we use $\mathcal{P}[\mathbf{D}, Q, \mathcal{D}]_i$ to denote $\mathcal{P}[Q, \mathcal{D}, D_i]$ where necessary to indicate the database history $\mathbf{D}$, query Q, and ranges $\mathcal{D}$.

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

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

Let us assume that the database before (after) the insertion of this tuple is D_1 (D_2), then we get:

$$\Delta D_2 = \{\mathbb{A}\ s_8\}$$

4.4 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 $\mathcal{R}$ over the domain of some attribute $a \in \mathbf{R}$, is a bag of pairs $\langle t, \mathcal{P}\rangle$ such that t is an m-ary tuple and $\mathcal{P} \subseteq \mathcal{R}$.*

We next define a operator $\mathbf{annotate}(R, \mathcal{R}, a)$ 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 $a \in \mathbf{R}$ and ranges $\mathcal{D} = \{\ldots, (\mathcal{R}, a), \ldots\}$, i.e., $(\mathcal{R}, a)$ is the partition for R in $\mathcal{D}$, the operator* **annotate** *returns a sketch-annotated relation $\mathcal{R}$ with the same schema as R:*

$$\mathbf{annotate}(R, \mathcal{D}) = \{\!\{\langle t, \{r\}\rangle \mid t \in R \wedge t.a \in r \wedge r \in \mathcal{R}\}\!\}$$

Given a database history, we define annotated deltas as database deltas where each tuple is annotated using the **annotate** operator. Consider ΔD_{ij} (the delta between D_i and D_j). Given a relation R and ranges $\mathcal{R}$ for attribute a, let R_i be the version of R in D_i. Then we define $\Delta\mathcal{R}_{ij}$ as:

$$\Delta\mathcal{R}_{ij} = \mathbf{annotate}(\Delta R_{ij}, \mathcal{R}, a)$$

Intuitively, $\Delta\mathcal{R}_{ij}$ contains all tuples from R that differ between D_i and D_j tagged with $\mathbb{A}$ or $\mathbb{D}$ depending on whether they got inserted or deleted and each such tuple t is paired with the range $r \in \mathcal{R}$ that $t.a$ belongs to. Analog we will use $\mathbf{annotate}(D, \mathcal{D})$.

Example 4.2. *Continuing with Ex. 4.1, the annotated version of ΔD_2 according to $\mathcal{R}_{price}$ is*

$$\{\!\{\langle \mathbb{A}\ s_8, \{r_3\}\rangle\}\!\},$$

because $s_8.price$ belongs to $r_3 = [1001, 1500] \in \mathcal{R}_{price}$.

4.5 Problem Definition

We are now ready to define **incremental maintenance procedures** that maintain provenance sketches. Such a procedure takes as input a query Q and an annotated delta $\Delta\mathcal{D}$ for the ranges $\mathcal{D}$ of a provenance sketch $\mathcal{P}$ and produces a delta $\Delta\mathcal{P}$ for the sketch. Incremental maintenance procedures are allowed to store some state $\mathcal{S}$ (e.g., information about groups produced by an aggregation operator) for a query Q and sketch $\mathcal{P}$ to allow for more efficient maintenance. Given the current state and $\Delta\mathcal{D}$, the maintenance procedure 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. As [30] demonstrated that any over-approximation of a safe provenance sketch is also safe, evaluating the query over the over-approximated sketch yields the same result as evaluating the query over the full database. Thus, a maintenance procedure always produces safe sketches.

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 $\mathcal{D}$. 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 $\Delta\mathcal{P}$:*

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

We require that for any Q, database D, delta ΔD, and provenance sketch $\mathcal{P}$ for Q and D, that

$$\mathcal{P}[Q, \mathcal{D}, D \cup \Delta D] \subseteq \mathcal{P} \uplus \Delta\mathcal{P}$$

5 INCREMENTAL ANNOTATED SEMANTICS

In this section, we will introduce an incremental maintenance procedure that maintains provenance sketches using annotated and incremental semantics for the operators of relational algebra. Under this semantics each operator takes as input an annotated delta produced by its inputs (or passed to the maintenance procedure 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 maintenance procedure. For an operator O (or query Q) we use $\mathcal{I}(O, \mathcal{D}, \Delta\mathcal{D}, \mathcal{S})$ ($\mathcal{I}(Q, \mathcal{D}, \Delta\mathcal{D}, \mathcal{S})$) to denote the result of evaluating O (Q) over the annotated delta $\Delta\mathcal{D}$ using the state $\mathcal{S}$. We will sometimes drop $\mathcal{S}$ and $\mathcal{D}$. Our incremental maintenance procedure evaluates a query expressed in relational algebra producing an updated state and outputting a delta where each row is annotated with a partial sketch delta. The procedure combines these partial sketch deltas to generate the final result $\Delta\mathcal{P}$.

5.1 Merging Sketch Deltas

As mentioned above, our incremental maintenance procedure evaluates the query using annotated, incremental versions of the operators used in the query and in a final step merges the sketch deltas for the query result to produce a delta for the current provenance sketch. 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 $\mathcal{S} : \mathcal{D} \to \mathbb{N}$ from the ranges to a counter for the number of result tuples that have a particular range in their sketch. If the counter for a fragment r reaches 0 (due to the deletion of tuples), then the fragment needs to be removed from the sketch. If the counter for a fragment r changes from 0 to a non-zero value, then the fragment now belongs to the sketch for the query and needs to be added to the previous version of the sketch (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 $\mathcal{S}'$ is computed and then explain how to compute $\Delta\mathcal{P}$ using the updated state $\mathcal{S}$. We define $\mathcal{S}'$ pointwise for a fragment r. Any newly inserted (deleted) tuple whose sketch includes r increases (decreases) the count for r. 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 r. Depending on the change of the count for r between $\mathcal{S}$ and $\mathcal{S}'$, the operator μ has to output a delta for $\mathcal{P}$. Specifically, if $\mathcal{S}[r] = 0 \neq \mathcal{S}'[r]$ then the fragment has to be inserted into the sketch and if $\mathcal{S}[r] \neq 0 = \mathcal{S}'[r]$ then the fragment was part of the sketch, but no longer contributes and needs to be removed.

$$\mathcal{S}'[r] = \mathcal{S}[r] + |\mathbb{\Delta}\ \mathcal{D}| - |\Delta\ \mathcal{D}|$$
$$\mathbb{\Delta}\ \mathcal{D} = \{\!\{\mathbb{\Delta}\ \langle t, \mathcal{P}\rangle^n \mid \mathbb{\Delta}\ \langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \wedge r \in \mathcal{P}\}\!\}$$
$$\Delta\ \mathcal{D} = \{\!\{\Delta\ \langle t, \mathcal{P}\rangle^n \mid \Delta\ \langle t, \mathcal{P}\rangle^n \in \mathcal{I}(Q, \Delta\mathcal{D}) \wedge r \in \mathcal{P}\}\!\}$$

$$\Delta\mathcal{P} = \bigcup_{r:\mathcal{S}[r]=0\wedge\mathcal{S}'[r]\neq 0} \{\mathbb{\Delta}\ r\} \cup \bigcup_{r:\mathcal{S}[r]\neq 0\wedge\mathcal{S}'[r]=0} \{\Delta\ r\}$$

Example 5.1. *Reconsider our running example from Ex. 1.2 that partitions based on $\mathcal{R}_{price}$. Assume that there are two result tuples t_1 and t_2 of a query Q that have fragment r_2 (based on range $r_2 = [601, 1000]$) in their sketch and one result tuple t_3 that has r_1 and r_2 in its sketch. Then the current state for the query is $\mathcal{P} = \{r_1, r_2\}$ and the state of μ is as shown below. If we are processing a delta $\Delta\ \langle t_3, \{r_1, r_2\}\rangle$ deleting tuple t_3, the updated counts $\mathcal{S}'$ are:*

$$\mathcal{S}[r_1] = 1 \qquad \mathcal{S}[r_2] = 3 \qquad \mathcal{S}'[r_1] = 0 \qquad \mathcal{S}'[r_2] = 2$$

As there is no longer any justification for r_1 to belong to the sketch (its count changed to 0), μ returns a delta: $\{\Delta\ r_1\}$

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 maintenance procedure unmodified. This operator does not maintain any state.

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

5.2.2 Projection. The projection operator also does not have to 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. The operator projects an input delta tuple on a list of expressions A. For each annotated delta tuple $\Delta\langle t, \mathcal{P}\rangle$, we project t on A and just propagate $\mathcal{P}$ unmodified as $t.A$ in the result depends on the same subset of the data as t in the input.

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

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 delta tuples that do not fulfill the selection condition.

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

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 the Q_1 with the current annotated state of Q_2 ($Q_2(\mathcal{D})$), (ii) joining the delta of the Q_2 with the current annotated state of Q_1 ($Q_1(\mathcal{D})$), (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 $s \circ t$ to compensate for the fact that two deletion deltas will be generated for $s \circ t$ through cases (i) and (ii). The non-annotated version of these rules have been discussed in [12, 16, 21, 24]. Finally, for $\mathbb{\Delta}\ s$ and $\Delta\ t$ and the symmetric case $\Delta\ s$ and $\mathbb{\Delta}\ t$, no output delta has to be produced as one of the join partners will not exist in the updated database.

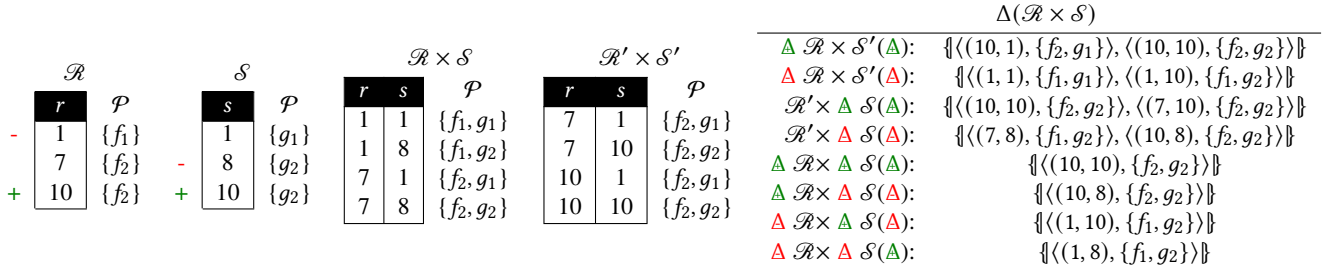


Figure 4: Incremental annotated semantics for cross product.

$$
\begin{aligned}
&\mathcal{I}(Q_1 \times Q_2, \Delta\mathscr{D}) = \\
&\{\!\{\mathbb{\Delta}\ \langle s \circ t, \mathcal{P}_1 \uplus \mathcal{P}_2\rangle^{n \cdot m} \mid \\
&\quad (\mathbb{\Delta}\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D}) \wedge \mathbb{\Delta}\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D})) \\
&\quad \vee (\mathbb{\Delta}\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D}) \wedge \Delta\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D})) \\
&\quad \vee (\mathbb{\Delta}\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D})) \wedge \langle t, \mathcal{P}_2\rangle^{m} \in Q_2(\mathscr{D})) \\
&\quad \vee (\langle s, \mathcal{P}_1\rangle^{n} \in Q_1(\mathscr{D})) \wedge \mathbb{\Delta}\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D}))\}\!\} \\
&\uplus \\
&\{\!\{\Delta\ \langle s \circ t, \mathcal{P}_1 \uplus \mathcal{P}_2\rangle^{n \cdot m} \mid \\
&\quad (\Delta\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D}) \wedge \mathbb{\Delta}\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D})) \\
&\quad \vee (\mathbb{\Delta}\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D}) \wedge \Delta\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D})) \\
&\quad \vee (\Delta\ \langle s, \mathcal{P}_1\rangle^{n} \in \mathcal{I}(Q_1, \Delta\mathscr{D})) \wedge \langle t, \mathcal{P}_2\rangle^{m} \in Q_2(\mathscr{D})) \\
&\quad \vee (\langle s, \mathcal{P}_1\rangle^{n} \in Q_1(\mathscr{D})) \wedge \Delta\ \langle t, \mathcal{P}_2\rangle^{m} \in \mathcal{I}(Q_2, \Delta\mathscr{D}))\}\!\}
\end{aligned}
$$

Example 5.2. *Fig. 4 shows annotated tables $\mathcal{R}$ and $\mathcal{S}$. Tuples with - were deleted and tuples with + were inserted. Consider the cross product of these tables before ($\mathcal{R} \times \mathcal{S}$) and after the update ($\mathcal{R}' \times \mathcal{S}'$) also shown in Fig. 4. $\Delta(\mathcal{R} \times \mathcal{S})$ in Fig. 4 shows all annotated delta tuples produced by incremental maintenance.*

5.2.5 Union. The union operator also does not require any state data. For a union $Q_1 \cup Q_2$, the incremental version of this operator propagates deltas from the input unmodified.

$$\mathcal{I}(Q_1 \cup Q_2, \Delta\mathscr{D}) = \mathcal{I}(Q_1, \Delta\mathscr{D}) \cup \mathcal{I}(Q_2, \Delta\mathscr{D})$$

5.2.6 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 an instance 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 wrt. to G.

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 $\mathscr{F}_g$ recording for each range r of $\mathcal{D}$ the number of input tuples belonging to the group with r in their provenance sketch. Intuitively, $\mathscr{F}_g$ 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. To simplify presentation, we will only formalize the case of a single aggregation function.

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

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

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

State Data Initialization. If the original database is not empty, then we can initialize the state based on the current database state as explained in the following. The state of the aggregation operator is initialized based on the current instance D. For each group $g \in \mathcal{G}(Q, D)$, we set

$$
\begin{aligned}
\mathcal{S}[g] &= (sum : s, cnt : c, ps : \mathcal{P}, map : \mathscr{F}_g) \\
s &= \sigma_{G=g}(\gamma_{\mathbf{sum}(a);G}(Q))(D) \\
c &= \sigma_{G=g}(\gamma_{\mathbf{count}(a);G}(Q))(D) \\
\mathcal{P} &= C(\sigma_{G=g}(\gamma_{\mathbf{sum}(a);G}(Q)), D, \mathcal{D}) \\
\forall r \in \mathcal{D} : \mathscr{F}_g[r] &= |\{\!\{t^n \mid t^n \in Q(D) \wedge r \in C(\sigma_{\text{SCH}(Q)=t}(Q), D, \mathcal{D})\}\!\}|
\end{aligned}
$$

That is, SUM and CNT are equal to the sum and count for the current group evaluated over D, $\mathcal{P}$ is the sketch computed for the query restricted to the group g, and the map $\mathscr{F}_g$ records how many tuples t in the result of Q have fragment r in their sketch. This can be computed by capturing a sketch for Q restricted to each $t \in Q(D)$ and counting tuples whose sketch contains r. Note that in our implementation we calculate this by running a single query that propagates sketches instead of evaluating a query for each $t \in Q(D)$.

Incremental Maintenance. The operator processes an annotated delta as explained in the following. Consider an aggregation $\gamma_{\mathbf{sum}(a);G}(Q)$ and annotated delta $\Delta\mathscr{D}$. Let ΔQ denote $\mathcal{I}(Q, \Delta\mathscr{D})$, i.e., the delta produced by incremental evaluation for Q using $\Delta\mathscr{D}$. 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 (count) are produced by adding $t.a \cdot n$ (n) for each inserted input tuple with multiplicity n: $\triangle \langle t, \mathcal{P}\rangle^n \in \Delta Q_g$ and subtracting this amount for each deleted tuple: $\triangle \langle t, \mathcal{P}\rangle^n \in \Delta Q_g$. For instance, if we are maintaining $\mathbf{sum}(A)$ and the delta contains the insertion of 3 duplicates of a tuple with A value 5, then the $\mathbf{sum}(A)$ should be increased by $3 \cdot 5$.

$$\text{CNT}' = \text{CNT} + \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} n - \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} n$$

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

The updated count in $\mathcal{F}'_g$ is computed for each $r \in \mathcal{D}$ as:

$$\mathcal{F}'_g[r] = \mathcal{F}_g[r] + \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge r \in \mathcal{P}} n - \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge r \in \mathcal{P}} n$$

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

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

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

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

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 $\triangle \langle g \circ (\text{SUM}'), \mathcal{P}'\rangle$ An existing group gets deleted if $\text{CNT} \neq 0$ and $\text{CNT}' = 0$. In this case we only output $\triangle \langle g \circ (\text{SUM}), \mathcal{P}\rangle$.

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'.

5.2.7 Aggregation: minimum and maximum. The aggregation functions **min** and **max** share the same state. For **min** (**max**), the key point is to get the next minimum value after update. To accomplish this, we use an ordered map to store all the values of aggregate attribute a. During the incremental maintenance, we insert all aggregate attribute value into the map for all inserted delta tuples and remove values from the map for all deleted annotated input tuples.

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

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

$\mathcal{P}$ and $\mathcal{F}_g$ are the same as described in aggregation function **sum** which store groups' sketch and each range the belong to this group with number of tuples containing this range. ORDMAP is an ordered map that record all values of aggregate attribute with the multiplicity, where the key of the map is the value of aggregate attribute and value of the map is the multiplicity of each aggregate attribute value. We use $\text{KEY}(\text{ORDMAP})$ to denote the all keys in the map ORDMAP, and use $\mathbb{M}(\text{ORDMAP})$ to denote the minimum key (maximum key for **max** aggregation function)in the map ORDMAP such that:

$$\mathbb{M}(\text{ORDMAP}) = \min(\text{KEY}(\text{ORDMAP})) = \min_{k \in \text{KEY}(\text{ORDMAP})} k$$

Incremental Maintenance. Consider an aggregation $\gamma_{\mathbf{min}(a);G}(Q)$ and annotated delta $\Delta\mathcal{D}$. Recall that ΔQ denotes the $\mathcal{I}(Q, \Delta\mathcal{D})$ and ΔQ_g to denote the subset of ΔQ including all annotated delta tuples $\Delta\langle t, \mathcal{P}\rangle$ where $t.G = g$. annotated delta tuples $\Delta\langle t, \mathcal{P}\rangle$ where $t.G = g$. We now discuss how to produce the output for one such group and how the incremental maintenance works for the case where the group already exists and still exits after maintenance.

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

$$\mathcal{S}[g] = (\text{ORDMAP}, \mathcal{P}, \mathcal{F}_g) \qquad \mathcal{S}'[g] = (\text{ORDMAP}', \mathcal{P}', \mathcal{F}'_g)$$

The ordered map ORDMAP is updated as follow: for each annotated tuple in ΔQ_g, the multiplicity of value in the map will be increase by 1 if it is an inserted annotated tuple, otherwise, the multiplicity will be decreased by one which the procedure is shown in algorithm 1. The updated count in $\mathcal{F}'_g$ is computed for each $r \in \mathcal{D}$ as:

Algorithm 1 Computing ORDMAP′

```
for Δ⟨t, P⟩ ∈ ΔQ_g do
    if Δ is △ then
        ORDMAP[t.a] ← ORDMAP[t.a] + 1
    else if Δ is ▲ then
        ORDMAP[t.a] ← ORDMAP[t.a] − 1
    end if
end for
```

$$\mathcal{F}'_g[r] = \mathcal{F}_g[r] + \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge r \in \mathcal{P}} n - \sum_{\triangle\langle t,\mathcal{P}\rangle^n \in \Delta Q_g \wedge r \in \mathcal{P}} n$$

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

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

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

$$\triangle \langle g \circ (\mathbb{M}(\text{ORDMAP})), \mathcal{P}\rangle \qquad \triangle \langle g \circ (\mathbb{M}(\text{ORDMAP}')), \mathcal{P}'\rangle$$

Creating and Deleting Groups. For groups g that are not in $\mathcal{S}$, we initialize the state for g as shown below: $\mathcal{S}'[g] = (\emptyset, \emptyset, \emptyset)$ and only output $\triangle \langle g \circ (\mathbb{M}(\text{ORDMAP}')), \mathcal{P}'\rangle$ An existing group gets deleted if $\exists k \in \text{KEY}(\text{ORDMAP}) : \text{ORDMAP}[k] > 0$ and $\forall k \in \text{KEY}(\text{ORDMAP}) : \text{ORDMAP}[k] = 0$. In this case we only output $\triangle \langle g \circ (\mathbb{M}(\text{ORDMAP})), \mathcal{P}\rangle$.

Max. For max, we maintain the same state as for min. The only difference is that we output the maximum key from the ordered map instead of minimum, which is the $\mathbb{M}(\text{ORDMAP})$ will get the maximum key from the map such that:

$$\mathbb{M}(\text{ORDMAP}) = \max(\text{KEY}(\text{ORDMAP})) = \max_{k \in \text{KEY}(\text{ORDMAP})} k$$

5.2.8 Top-K. Top-K operator returns the first k tuples from all input tuples according the order attributes in O. To maintain this operator, the strategy is: first we output first k annotated tuples from current state as deleted output, then we update the state based on annotated delta input and last we propagate the top k annotated tuples as inserted output from the updated state. To fast access to the first k smallest tuples among all tuples, we use an ordered map as state to store all tuples comes to this operator when building the state.

State Data. Consider an top-k operator $\tau_{k,o}(Q)$. To successfully maintain the result and provenance sketch, we store the following state:

$$\mathcal{S} = (\text{ORDMAP})$$

The ORDMAP is an ordered map, in which each key is a distinct order value and value is a map where map's key is annotated tuple and map's value is the multiplicity of the annotated tuple.

Updating State. Assume the current and updated state is shown below:

$$\mathcal{S} = (\text{ORDMAP}) \qquad \mathcal{S}' = (\text{ORDMAP}')$$

To update the state, we need to check the ordered value for each delta annotated tuple such that

$$\mathcal{S}'[o][\langle t,\mathcal{P}\rangle] = \mathcal{S}[o][\langle t,\mathcal{P}\rangle] + \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q \wedge t.o=O} n \; - \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q \wedge t.o=O} n$$

Recall KEY(m) return the key set of map m. We now denote $\mathbb{E}(()\cdot)$ a function takes an ordered map of top-k state and returns all annotated tuple with the multiplicity stored in this map in ascending order by O such that:

$$\mathbb{E}(\text{ORDMAP}) = \{\!\{ \bigcup \langle t,\mathcal{P}\rangle^n \mid o \in \text{KEY}(\text{ORDMAP}) \wedge \langle t,\mathcal{P}\rangle \in \text{ORDMAP}[o] \wedge n = \text{ORDMAP}[o][\langle t,\mathcal{P}\rangle] \}\!\}$$

We then output annotated tuples as the output of maintenance:

$$\mathbb{A}\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP})) \qquad \mathbb{A}\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}'))$$

Inserting a new entry. For a inserted annotated tuple, if the ORDMAP does not contain the key (order-by value), then we create an entry for this value, where value of this key of the ordered map is another map. We set the annotated tuple as key in the inner map with a multiplicity 1. If the state map contains the entry but inner map does contain the key, we just add a key-value pair into the inner map.

6 CORRECTNESS PROOF

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

THEOREM 6.1 (CORRECTNESS). *$\mathcal{I}$ as defined in this section is an incremental maintenance procedure.*

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. 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 [**?**]. □

7 THE IMP SYSTEM

While the semantics from Sec. 5 can be implemented in SQL (discussion in [**?**]), an in-memory implementation can be significantly more efficient as we can utilize data structures that are not available to us in a SQL-based implementation. Furthermore, as deltas are typically small, having a purely in-memory engine is typically not a problem. Thus, we implemented IMP 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 $\mathcal{I}(Q,\mathcal{S},\Delta D)$ to generate delta sketches. For joins (and cross products), $\Delta\mathscr{R} \bowtie \mathscr{S}$ and $\mathscr{R} \bowtie \Delta\mathscr{S}$ are executed by sending $\Delta\mathscr{R}$ ($\Delta\mathscr{S}$) to the database and evaluating the join in the database.

7.1 Storage Layout & State Data

All input and output data of an operator is stored in a columnar representation for horizontal chunks of a table (*data chunks*) with a maximum number of row. 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 [30] 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 (a snapshot identifier for databases that use snapshot isolation) to record which database version the sketch corresponds to. IMP can persist its state in the database.

Aggregation and Top-k. For aggregation, we use hashmaps to implement the state for **sum**, **count**, and **avg**. For **min** and **max** we use red-black trees. The aggregation result for a single group may be updated multiple times when processing a delta with multiple tuples. To avoid producing multiple delta tuples per group we maintain copies of the previous state's of groups before an incremental maintenance operations that are created lazily when a group is updated for the first time when processing a delta $\Delta\mathscr{D}$. Once a delta has been processed, we use the per batch data structure to determine which groups have been updated and output deltas for the group as described in Sec. 5.2.6 and 5.2.7. We follow the same approach for the top-k operator.

7.2 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 client uses the database engine to compute the result of $\mathscr{R} \bowtie \Delta\mathscr{S}$ and $\Delta\mathscr{R} \bowtie \mathscr{S}$ by sending $\Delta\mathscr{R}$ (or $\Delta\mathscr{S}$) to the database and evaluate the join. IMP maintains bloom filters on the join attributes for both sides of equi-joins that are used to filter out rows from $\Delta\mathscr{R}$ (and $\Delta\mathscr{S}$) that do not have any join partners in the other table. If according to bloom filter no rows from the delta have join partners then we can avoid the round trip to the database completely.

Filtering Deltas Based On Selections. If a query involves selections 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 result of incremental maintenance as their decedents will be filtered by the incremental version of selection. That is, we can push the selection conditions into the query that retrieves the delta from the database.

Optimizing Minimum, Maximum, and Top-k. If the input to an aggregation with **min** or **max** or the top-k is large, then maintaining the sorted map can become a bottleneck. Instead of storing the full input in the sorted map, we can instead only store the top / bottom m tuples. By keep a record of the first m tuples, it is safe-guaranteed to delete m tuples from its input. This is useful when dealing with deletion. For example, if we keep first 20 minimum values for a group, from the input tuples of this group, the state data can support deletions that removes no greater than 20 tuples. To achieve this, we pass an parameter to IMP engine, and when building the state data for these operators, the engine will only get a certain number of tuples to build the state. This optimization will help for deletion, because it is necessary to know the next minimum/maximum, while for insertion, only one tuple per group stored can make incremental maintenance work, because the only tuple is always the current minimum/maximum and the new minimum/maximum will always be in the state data if it comes from inputs.

7.3 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 use techniques like reenactment [3, 4] to reconstruct such database states.

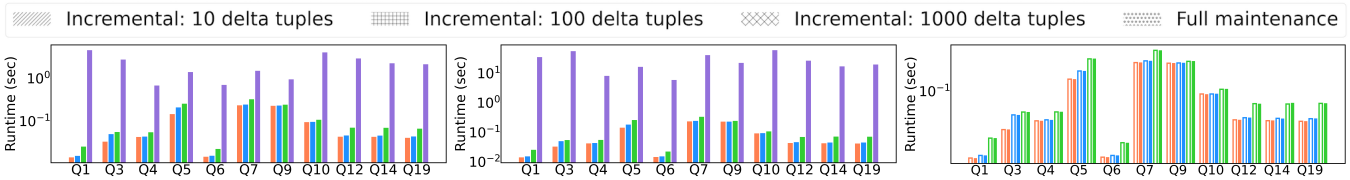
7.3.1 Computing Delta Data. The IMP will use delta data to compute the delta provenance sketch for queries. Each time a statement modifies the database, we record the tuples affected in database: for insertion and deletion, we just keep a copy of these inserted or deleted tuples, and for tuples updated, we keep two copies of both original and current tuples. For example, for a statement `UPDATE R SET A = 3 WHERE B < 4`, two copies of tuples will be generated: before update attribute `A` and a copy of tuples `A = 3` for all those `B < 4`.

7.3.2 Transactions. A transaction is collection of work that changes the state of a database. For a transaction, it will see the same snapshot isolation within its own update. If sketch maintenance is executed as part of this transaction, then the maintained sketch is based on the snapshot the delta data generated.

While snapshot isolation just specifies the data read from the database and there are no locks placed on that data. Thus, snapshot isolation does not block other transactions from writing data. Therefore, there is a problem: when a transaction (T1) including a maintenance procedure and an execution of query using provenance sketches is executing, another simultaneous transaction (T2) including modifying a relation on which the provenance sketch is captured comes in and executes after maintenance procedure but before query running of T1. Therefore, the result of running the query may be not correct due to transaction T2's modification to the database and the sketch is not up-to-date.

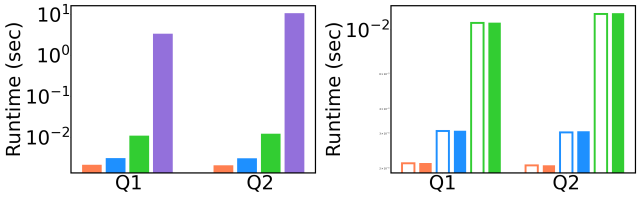
There are several ways to solve this challenge. The first one is to use lock. For each transaction execution, locks are places on the data the transaction accesses. Before this transaction completes, no other transactions can write to the data. In this case, there is no modification to database except the transaction itself and it ensures that if sketch maintenance part of this transaction, the updated sketches are up-to-date.

Another way is statement level snapshot isolation. In this way, for each sketch, we set a version number (like SCN in Oracle database) to reflect on which version of the database it is created. If the sketches need to be updated, the delta data is the difference between two consecutive version of the database. If a query runs with sketches, it is required to ensure that the query runs with sketches having the same snapshot version number as the database does. This ensures the sketches correctly reflect relevance among queries and the database. While, there is a question for the statement level snapshot isolation: each query acquires its own version. For this question, we can simply relax the version number of query, which means that no matter what the query version number is, we treat it as if it has the same version as the sketches has because for a query execution with sketches, it will return the correct result of a certain state of database if the sketches are correct based on that database. Thus, for statement level snapshot isolation, if to speed-up queries execution with sketches and database changing(two versions: v1 followed by v2), first, maintain sketches to version v2 using delta data from the difference between v1 and v2 of the database, and then apply sketches and database of v2.



(a) TPC-H 1GB (b) TPC-H 10GB (c) TPC-H 10GB Insert and Delete

Figure 5: TPC-H Benchmark



(a) Crimes (b) Insertion and deletion

Figure 6: Crimes

7.4 Cost Estimation

[30] has studied the cost of capture provenance sketches and benefits to use sketches to answer queries. In this work, we will have a brief discuss about a linear model for cost estimation for.

We consider one query Q and a database D. Suppose there are n+1 versions of databases with each one D_i having delta ΔD_i between D_i and D_{i-1}, the provenance sketch $\mathcal{P}_i$. The query Q executes multiple times defined as ${R^{Q^{\#}}}_i$. At the version D_i, if we execute the query Q, we can have the following two ways: 1. simply executing Q over D_i; 2. using provenance sketch $\mathcal{P}_i$ to answer Q. If we just execute Q over D_i (no sketches used), we donate the runtime as $C^{noPS}(Q, D_i)$, and the total cost of query Q at database version D_i will be: $C^{noPS}(Q, D_i) \times {R^{Q^{\#}}}_i$. To use the sketch $\mathcal{P}_i$ to answer this query, we specify into two approaches of provenance maintenance: incremental and full maintenance. If we apply full maintenance approach to update the sketch, we need to capture the sketch from scratch since the database has been updated. We defined the cost as $C^{cap}(Q, D_i)$. Then we use the sketch $\mathcal{P}_i$ to answer the query Q and the time cost is defined as $C^{use}(Q, D_i, \mathcal{P}_i)$. The total cost of query Q executed multiple time when using full maintenance to maintain sketch at D_i will be: $C^{cap}(Q, D_i) + C^{use}(Q, D_i, \mathcal{P}_i) \times {R^{Q^{\#}}}_i$. If the sketch is maintained incrementally, the time to update the sketch is defined as $C^{maintain}(Q, D_{i-1}, \Delta D_i)$. Then we use $\mathcal{P}_i$ to answer Q. In this case, the total time cost will be: $C^{maintain}(Q, D_{i-1}, \Delta D_i) + C^{use}(Q, D_i, \mathcal{P}_i) \times {R^{Q^{\#}}}_i$. Note that if the query is first encountered, the incremental approach is not applicable and the time cost is the same as full maintenance approach.

[30] has studied the cost of capture provenance sketches and benefits to use sketches to answer queries. In this work, we will have a brief discuss about a linear model for cost estimation for.

We consider one query Q and a database D. Suppose there are n+1 versions of databases with each one D_i having delta ΔD_i between D_i and D_{i-1}, the provenance sketch $\mathcal{P}_i$ and total ${R^{Q^{\#}}}_i$ times running a query Q, as shown below.

	$\cdots$	T_n			
$Database:$	D_0	$\cdots$	D_i	$\cdots$	D_n
$Delta:$	ΔD_0	$\cdots$	ΔD_i	$\cdots$	ΔD_n
$ProvenanceSketch:$	$\mathcal{P}_0$	$\cdots$	$\mathcal{P}_i$	$\cdots$	$\mathcal{P}_n$
$TimesOfQueryExecution:$	${R^{Q^{\#}}}_0$	$\cdots$	${R^{Q^{\#}}}_i$	$\cdots$	${R^{Q^{\#}}}_n$

At the version of D_i point, if the query Q needs to be executed, there are mainly two different ways: using sketch or not using sketch. For queries running without sketches, we denote $C^{noPS}(Q, D_i)$ to be the time cost for a query running over database D_i. If using sketch to answer Q, we have two approaches: first one is that each time we capture provenance sketches and then use the sketch when queries running. The other one is to update sketches from old ones based on deltas between the database change and then apply the sketches when running the queries. Note that for the later approach that queries run with sketches, there must exist previous provenance sketches otherwise the first task of this approach is to capture the sketches. We denote $C^{cap}(Q, D_i)$ to be the cost for capturing sketches for a query Q over the database D_i at time point T_i, $C^{maintain}(Q, D_{i-1}, \Delta D_i)$ to be the cost for maintaining sketches for a query Q according to the state data from database D_{i-1} and delta information ΔD_i and $C^{use}(Q, D_i, \mathcal{P}_i)$ to be the time cost for a query Q over the database D_i using the sketch $\mathcal{P}_i$. Then, at time point T_i, we cat build formula to get cost of running with different approaches. We define T_i^{noPS} as running without provenance sketches for multiple times, $T_i^{m\&uPS}$ as first maintaining sketches and then run queries with sketches for several times and $T_i^{c\&uPS}$ as capturing sketches and then running with sketches. Then we can get each time cost of different running types as following:

$$\begin{cases} T_i^{noPS} = C^{noPS}(Q, D_i) \times R^{Q^{\#}} \\ T_i^{c\&uPS} = C^{cap}(Q, D_i) + C^{use}(Q, D_i, \mathcal{P}_i) \times R^{Q^{\#}} \\ T_i^{m\&uPS} = C^{maintain}(Q, D_{i-1}, \Delta D_i) + C^{use}(Q, D_i, \mathcal{P}_i) \times R^{Q^{\#}} \end{cases}$$

For a series of n updates to the database, we can get the total total cost of all execution of query Q.

$$\begin{cases} TOT^{noPS} = \sum_{i=0}^{n} T_i^{noPS} \\ TOT^{c\&uPS} = \sum_{i=0}^{n} T_i^{c\&uPS} \\ TOT^{m\&uPS} = T_0^{c\&uPS} + \sum_{i=1}^{n} T_i^{m\&uPS} \end{cases}$$

8 EXPERIMENTS

In this section, we discuss experiments conducted to 1. study how IMP's runtime is affected by the characteristics of a workload such as the database size, size of the delta, and query structure; 2. compare our incremental maintenance against full maintenance (capturing provenance sketches from scratch); and 3. evaluate the effectiveness of the proposed optimizations. 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). The database backend for all experiments was Postgres 16.2. We repeated each experiment at least 10 times and report median runtime. The variance of runtimes is low, within a few percent.

8.1 Datasets and Workloads

We use the TPC-H benchmark, a real world Crime dataset and synthetically generated data for microbenchmarks. For synthetic data, we generated several tables with 10M rows where each one contains at least 11 attributes with different datatypes. Each synthetic table has a key attribute id. For the other attributes, we generate the values of one attribute (a) by sampling uniformly from a given domain. Then, we generate the remainder of the attributes to be linearly correlated with a subject to Gaussian noise to create partially correlated values.

Baselines. For sketch maintenance, we evaluate the performance of IMP (*incremental maintenance*) against a baseline where sketches are captured from scratch (*full maintenance*). Furthermore, in some experiments we compare against a baseline (*non-sketch*) that does not use provenance-based data skipping.

Parameter. We vary the *delta size* (the number of tuples that are inserted or deleted) in all experiments: (i) mostly we use realistic delta sizes: 10, 50, 100, 500 and 1000; (ii) for determining the break-even point between incremental and full maintenance we vary the the delta size from 0.1% of the table to 10% in 0.1% increments.

8.2 TPC-H

We use the TPC-H (www.https://www.tpc.org/tpch/) benchmark at SF1 (~ 1 GB) and SF10 (~ 10GB) to compare incremental against full maintenance using delta sizes from 10 to 1000 tuples. We selected queries that benefit from provenance sketches [30] and are sufficiently complex (multiple joins, aggregation with `HAVING` or top-k). In these experiments, we turn on the selection push down and join bloom filter optimizations (see Sec. 7.2). Fig. 5a and Fig. 5b show the runtime of incremental vs. full maintenance for SF1 and SF10, respectively. Note that the runtime of full maintenance does only depend on the current size of the database, we do not show results for different delta sizes for this method. Incremental maintenance outperforms full maintenance by at least a factor of 3.9 and up to a factor of 2497, demonstrating the effectiveness of incremental maintenance. As joins require round trips to the database and evaluation of queries, the number of joins affects the performance of IMP. Nonetheless, even for TPC-H $Q5$, $Q7$ and $Q9$ that contain 6-way joins, IMP still outperforms full maintenance by about 2 orders of magnitude for SF10. Importantly, the runtime of IMP is mostly unaffected by the size of the database as the runtime of most incremental operators only depends on the size of the delta.

Fig. 5c shows the incremental maintenance runtime for both insertion and deletion for certain amount of delta for 10 GB database size.

8.3 Crime Dataset

The Crime[3] dataset records incidents in Chicago and consists of a single 1.87GB table with $7.3M$ rows. We use two queries (SQL code for all queries is shown in [23]): `CQ1`: The numbers crime of each year in each beat (geographical location). `CQ2`: Areas with more than 1000 crimes. We use realistic delta sizes (10 to 1000). As shown in Fig. 6a incremental maintenance outperform full maintenance by at least 2 orders of magnitude. Fig. 6b show the incremental maintenance runtime for both insertion and deletion under given delta size.

8.4 Mixed Workloads

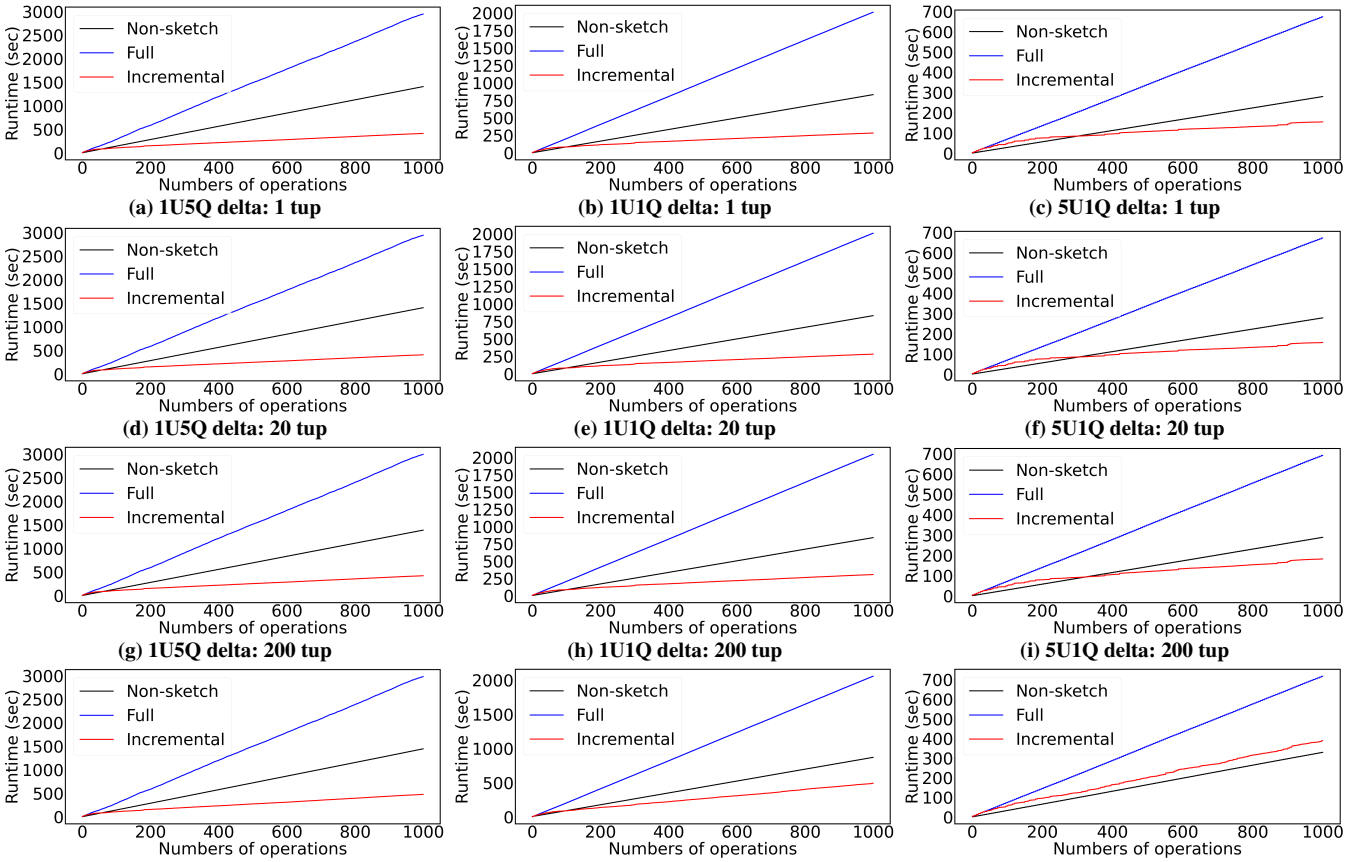
In this experiment, we evaluate our approach on mixed workloads consisting of queries and updates and measure the end-to-end runtime. Each workload consists of 1000 operations. We control the ratio between queries and updates referred to as the *query-update ratio* and compare against full maintenance and an approach that does not use provenance sketches (non-sketch). Sketch-based methods start with an empty set of sketches. We use the technique from [30] 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 for the current query. When a update is executed, we determine the delta and append it to the delta table that we maintain for every table. The delta tables cache deltas. Tuples in the delta tables are associated with a snapshot identifier that enables us to fetch only delta tuples of updates that were executed after the sketch was last maintained.

For this experiment, we utilize a query template $Q_{endtoend}$ 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, we always recapture sketches to answer the queries when encountering queries. Fig. 7 shows the runtime for several combinations of query-update ratio and delta size (we present additional combinations in [23]). Full maintenance approach has the highest cost, because the cost incurred by recapturing sketches frequently outweighs the benefit of using these sketches to speed-up queries. IMP outperforms both baselines, except for the extreme case 5U1Q with delta size 2000 (per update) where 5 updates, each affecting at least 10000 tuples are executed between two adjacent queries, since the adjacent queries may not use the same sketch.

8.5 Microbenchmarks

In this set of experiments, we use synthetically generated data and vary specific aspects of queries to evaluate in detail how our approach is affected. We compare the performance of incremental maintenance against full maintenance as a baseline varying the delta size. The database size is kept constant, i.e., for full maintenance, the runtime is not affected by varying the delta size. For incremental

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



(a) 1U5Q delta: 1 tup (b) 1U1Q delta: 1 tup (c) 5U1Q delta: 1 tup

(d) 1U5Q delta: 20 tup (e) 1U1Q delta: 20 tup (f) 5U1Q delta: 20 tup

(g) 1U5Q delta: 200 tup (h) 1U1Q delta: 200 tup (i) 5U1Q delta: 200 tup

(j) 1U5Q delta: 2000 tup (k) 1U1Q delta: 2000 tup (l) 5U1Q delta: 2000 tup

Figure 7: End to end experiments

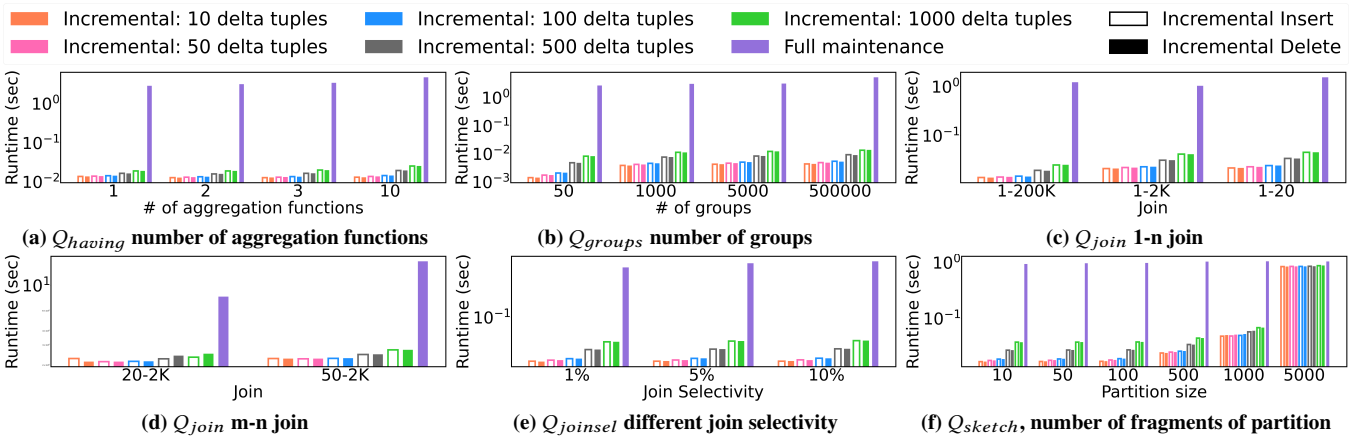
maintenance, we examine consider both realistic delta sizes (Fig. 8) and use large deltas to determine the "break even" point: for delta sizes larger than that, full maintenance outperforms incremental maintenance. For smaller deltas we test both insertions and deletions.

Number of groups. We use a query Q_{groups} (SQL code shown in [23]) that is group-by aggregation with `HAVING` over a single table and vary the number of groups: 50, $1K$, $5K$ and $500K$. As the data structures maintained by our approach for aggregation stores an entry for each group and the number of groups affected by a delta also depends on the number of groups, we expect that runtime will increase when increasing the number of groups. As shown in Fig. 8b, for delta sizes up to 1000 tuples, incremental maintenance outperforms full maintenance by 2 (500k groups) to 3 (50 groups) orders of magnitude. Fig. 9b shows that the break even point (where full maintenance starts to outperform incremental maintenance) lies at delta sizes between ~3.5% (for 50 group) and ~ 5.5% for (500k groups). While the runtime of incremental maintenance increases when increasing the number of groups, the effect is more pronounced for full maintenance that has to calculate results for all groups.

Number of aggregation functions. In this experiment, we use a query Q_{having} (see [23]) which is an group-by aggregation on a single table with filtering in the `HAVING` clause on the aggregation function result. The total number of groups is set to 5000 in this experiment. We vary the number of aggregation functions used in the `HAVING` condition as our approach has to maintain the results for these aggregation. Fig. 8a and Fig. 9a show the runtime of incremental vs. full maintenance using both realistic delta sizes and large deltas. The runtime of incremental maintenance is linear in the size of the delta for this query with a coefficient that depends on the number of aggregation functions. For realistic delta sizes, incremental maintenance outperforms full maintenance by ~ 2 orders of magnitude. As shown in Fig. 9a, incremental maintenance is faster than full maintenance for deltas of up to ~ 5% of the database.

Joins. We evaluate the performance of group-by aggregation queries with `HAVING` over the result of an equi-join using query template Q_{join} (see [23]). 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.

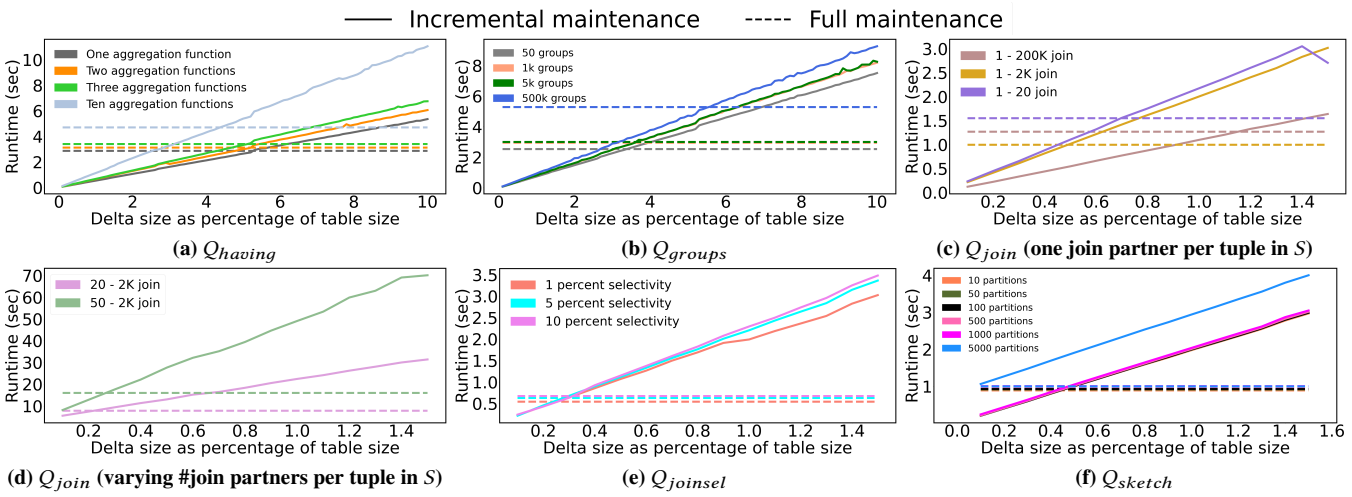
Fig. 8c and Fig. 9c show the runtime of incremental vs. full maintenance for $1 - n$ joins, and Fig. 8d and Fig. 9d show the performance of both full and incremental approaches for $m - 2K$ joins. In the 1-n join experiment, the 1-20 join is more expensive



(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) $Q_{joinsel}$ different 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}

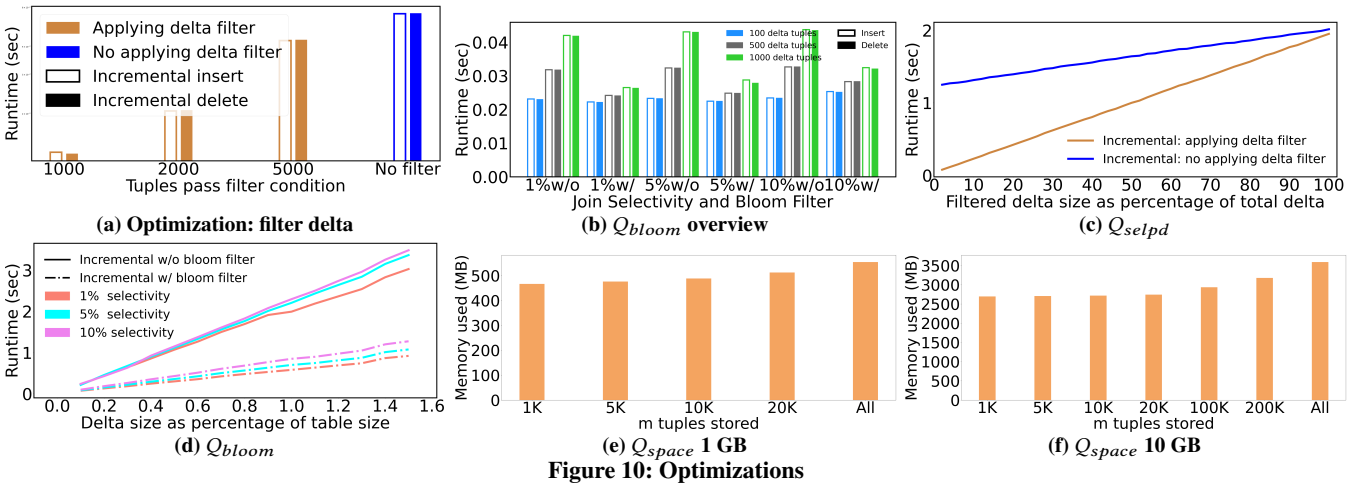
Figure 9: Microbenchmarks: varying delta size to determine the "break even point" where full maintenance outperforms IMP.

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. 8.2, 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. 8.6.

Join selectivity. To evaluate performance of queries with more selective joins, we use query template group-by-aggregation over join: $Q_{joinsel}$ (R join S) (see [23]), to evaluate different join selectivity: 1%, 5%, and 10%. We control the join attribute values in table S (this attribute joins with another attribute in table R) such that only certain percentage of tuples join. Fig. 8e and Fig. 9e show the runtime of incremental and full maintenance using both realistic delta size and large deltas. For small deltas, the selectivity of the join has less of an impact on incremental maintenance than for large delta sizes. This is due to the fact that for small deltas we are joining a small table ($\Delta\mathscr{R}$) with a large table ($\mathscr{R}S$), i.e., the bottleneck is scanning the large table.

Varying Partition Granularity. In this experiment, we vary #frag, the number of fragments of the partition on which provenance sketches are based on. We use query template Q_{sketch} (SQL code shown in [23]) which is an 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 incremental and full maintenance for both realistic and large deltas. While the cost of full maintenance is impacted #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 number of tuples in the delta



(a) Optimization: filter delta

(b) Q_{bloom} overview

(c) Q_{selpd}

(d) Q_{bloom}

(e) Q_{space} 1 GB

(f) Q_{space} 10 GB

Figure 10: Optimizations

and the cost paid per tuple is roughly linear in #frag as intermediate sketches of large size have to be processed.

8.6 Optimizations

To understand the impact of our optimizations, we use the synthetic dataset to evaluate the performance of IMP on both realistic delta sizes and large deltas. We only evaluate the performance of the incremental approach.

Filtering deltas inputs based on selection conditions. In this experiment, we evaluate the effectiveness of our delta selection push-down optimization that pre-filters the delta based on selection conditions in the query. We use the query Q_{selpd} which is a group-by aggregation query without joins (SQL code is shown in [23]). We evaluate the performance of incremental maintenance with and without delta filtering, varying the selectivity of the query's selection condition (`WHERE` clause). We fix the delta size to 2.5% of the table. Then we gradually increase the fraction of the delta that fulfills the selection conditions from 2% to 100%. Fig. 10c shows results of this experiment. The results show that the runtime of filtering deltas increases linearly in the selectivity of the query's selection condition. Even for larger selectivities, the cost of filtering delta tuples is amortized by reducing the cost of incremental maintenance (and communication between IMP and that database). Thus, this optimization should be applied whenever possible.

Join optimization using bloom filters. Another optimization we applied in IMP engine is to use bloom filters for each join to track which tuples potentially have join partners. This enables us to avoid evaluating a join $\Delta\mathcal{R} \bowtie \mathcal{S}$ ($\mathcal{R} \bowtie \Delta\mathcal{S}$) using the database when we can determine based on the bloom filter that no tuples in the delta have any join partners or at least reduce the delta size. To evaluate this optimization, we use again query $Q_{joinsel}$ (see [23]) which is a group-by aggregation with `HAVING` over the result of a join. Fig. 10b shows the runtime of incremental maintenance applying bloom filter for joins varying selectivity selectivity and delta size. Fig. 10d shows runtime of incremental maintenance for large delta sizes. The result shows that filtering the delta using bloom filters is effective 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. As show in Fig. 10d, bloom filter optimization is effective for both low and high selectivity and across all delta sizes we have tested.

Memory space optimization. We evaluate the main memory optimization for data structures used for **min**, **max** and top-k operators to store only top m tuples. In this experiment, we use a query Q_{space} (TPC-H Q10) as our example. For 1 GB dataset, the total tuples number for top-k is 37293 and for 10 GB dataset, the number is 371104. We vary the number for m to examine how much memory used (MB) if we store m tuples varying the number in the data structure. For TPC-H 1GB, we vary the number of m in $1K$, $5K$, $10K$, $20K$ and all tuples. For TPC-H 10GB, these number values are $1K$, $5K$, $10K$, $20K$, $100K$, $200K$ and all. Fig. 10e and Fig. 10f show the memory used varying the stored tuple number m. A knowledge learned from the experiments is that memory saving can be achieved by reducing the number of tuples kept in the state data structure.

8.7 Summary

Our incremental approach outperforms the full maintenance method in all cases for realistic delta sizes and in many cases for large delta sizes. The performance difference is often several orders of magnitude. While the data transfer, fetching a delta from the database or sending a delta to the database, to evaluate the incremental part of a join (e.g., $\Delta\mathcal{R} \bowtie \mathcal{S}$) is a major factor in the runtime of IMP, our optimizations are effective in reducing data transfer. Our experiments demonstrate that our incremental maintenance engine scales well with respect to database size as the runtime for many incremental operators only depends on the size of the delta and is independent of the size of the database.

9 CONCLUSIONS AND FUTURE WORK

We present the first approach for in-memory incremental maintenance of provenance sketches (IMP), an approach to maintain provenance sketches incrementally under database updates. Our in-memory engine for sketch-annotated data can efficiently produce updates to sketches to correctly reflect which parts of the updated database contain provenance information that is needed to answer

a query and, thus, should belong to the sketch. Using bloom filters and selection-push-down, we further improve the performance of the incremental maintenance process. 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 operations and nested or recursive queries, we will investigate how to integrate provenance-based data skipping and incremental maintenance of sketches with cost-based query optimization and self-tuning.

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

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

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

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

[23] Pengyuan Li, Boris Glavic, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic, and Xing Niu. [n.d.]. In-memory Incremental Maintenance of Provenance Sketches (full version). https://github.com/IITDBGroup/2025_PVLDB_IMP/blob/main/imp_techreport.pdf.

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

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

[26] Haneen Mohammed, Charlie Summers, Sughosh Kaushik, and Eugene Wu. 2023. SmokedDuck Demonstration: SQLStepper. In *SIGMOD*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). 183–186.

[27] 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.

[28] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

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

[37] 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, October 31 - November 3, 1994, Toronto, Ontario, Canada*. 70.

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

[39] 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.

[40] 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.

A CORRECTNESS PROOF

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

THEOREM A.1 (CORRECTNESS). *$\mathcal{I}$ as defined in this section is an incremental maintenance procedure.*

In this section, we will show the correctness of incremental operator semantics in two aspects: 1. tuple correctness (**tuple correctness**), in which the incremental maintenance procedure can always generate the correct tuple set for the operators it maintains, 2. fragment correctness (**fragment correctness**), in which the maintenance procedure can output the correct delta sketches as well for the operators it maintain.

We denote Q^n to be a query having at most n operators and $\mathbb{Q}^i$ to be the class of all queries with i operators such $Q^i \in \mathbb{Q}^i$. Given a database $D = \{\!\{t_1, \ldots, t_n\}\!\}$ and the annotated database $\mathcal{D} = \{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_n, \mathcal{P}_n\rangle\}\!\}$. Suppose the result of running query Q over the database D is

$$Q(D) = \{\!\{t_{o_1}, \ldots, t_{o_m}\}\!\}$$

and the annotated result is

$$Q(\mathscr{D}) = \{\!\{\langle t_{o_1}, \mathcal{P}_{o_1}\rangle, \ldots, \langle t_{o_m}, \mathcal{P}_{o_m}\rangle\}\!\}$$

Given the delta database $\Delta D = \mathbb{A}D \uplus \mathbb{\Delta}D$ where $\mathbb{A}D = \{\!\{t_{i_1}, \ldots, t_{i_k}\}\!\}$ and $\mathbb{\Delta}D = \{\!\{t_{d_1}, \ldots, t_{d_x}\}\!\}$. Let D' to be the updated database such that:

$$D' = D \uplus \Delta D$$

Suppose the result of running query Q over the updated database is

$$Q(D') = \{\!\{t_{o_1}, \ldots, t_{o_l}\}\!\}$$

and the annotated result is

$$Q(\mathscr{D}') = \{\!\{\langle t_{o_1}, \mathcal{P}_{o_1}\rangle, \ldots, \langle t_{o_l}, \mathcal{P}_{o_l}\rangle\}\!\}$$

We define $\mathbb{T}(\cdot)$ (**extract tuples**) is a function that takes a bag of annotated tuples and return all the tuples from the bag such that:

$$\mathbb{T}(\{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_y, \mathcal{P}_y\rangle\}\!\}) = \{\!\{t_1, \ldots, t_y\}\!\}$$

Lemma A.2. *Let $\mathbb{T}(\cdot)$ be the **extract tuples** function, $\mathscr{D}_1$ and $\mathscr{D}_2$ be two annotated databases. The following property holds:*

$$\mathbb{T}(\mathscr{D}_1 \cup \mathscr{D}_2) = \mathbb{T}(\mathscr{D}_1) \cup \mathbb{T}(\mathscr{D}_2)$$

Proof. Suppose $\mathscr{D}_1 = \{\!\{\langle t_1, \mathcal{P}_{t_1}\rangle, \ldots, \langle t_m, \mathcal{P}_{t_m}\rangle\}\!\}$ and $\mathscr{D}_2 = \{\!\{\langle s_1, \mathcal{P}_{s_n}\rangle, \ldots, \langle s_n, \mathcal{P}_{s_n}\rangle\}\!\}$. Then $\mathscr{D}_1 \cup \mathscr{D}_2$, $\mathbb{T}(\mathscr{D}_1)$ and $\mathbb{T}(\mathscr{D}_2)$ are:

$$\mathscr{D}_1 \cup \mathscr{D}_2 = \{\!\{\langle t_1, \mathcal{P}_{t_1}\rangle, \ldots, \langle t_n, \mathcal{P}_{t_n}\rangle, \langle s_1, \mathcal{P}_{s_1}\rangle, \ldots, \langle s_n, \mathcal{P}_{s_n}\rangle\}\!\}$$
$$\mathbb{T}(\mathscr{D}_1) = \{\!\{t_1, \ldots, t_m\}\!\} \qquad \mathbb{T}(\mathscr{D}_2) = \{\!\{s_2, \ldots, s_n\}\!\}$$

We can that $\mathbb{T}(\mathscr{D}_1 \cup \mathscr{D}_2)$ is:

$$\mathbb{T}(\mathscr{D}_1 \cup \mathscr{D}_2) = \{\!\{t_1, \ldots, t_m, s_1, \ldots, s_n\}\!\}$$

and $\mathbb{T}(\mathscr{D}_1) \cup \mathbb{T}(\mathscr{D}_2)$ is:

$$\mathbb{T}(\mathscr{D}_1) \cup \mathbb{T}(\mathscr{D}_2) = \{\!\{t_1, \ldots, t_m\}\!\} \cup \{\!\{t_m, s_1, \ldots, s_n\}\!\} = \{\!\{t_1, \ldots, t_m, s_1, \ldots, s_n\}\!\}$$

Therefore, $\mathbb{T}(\mathscr{D}_1 \cup \mathscr{D}_2) = \mathbb{T}(\mathscr{D}_1) \cup \mathbb{T}(\mathscr{D}_2)$ □

From Lemma A.3, we can know that $\mathbb{T}(\Delta\mathscr{D}) = \mathbb{T}(\mathbb{A}D) \cup \mathbb{T}(\mathbb{\Delta}D)$, since $\Delta\mathscr{D} = \mathbb{A}\,\mathscr{D} \cup \mathbb{\Delta}\,\mathscr{D}$

Lemma A.3. *Like lemma A.3, we can have that: $\mathbb{T}(\mathscr{D}_1 - \mathscr{D}_2) = \mathbb{T}(\mathscr{D}_1) - \mathbb{T}(\mathscr{D}_2)$*

Lemma A.4. *Let $\mathbb{T}(\cdot)$ be the **extract tuples** function, $\mathscr{D}$ and $\Delta\mathscr{D}$ be two annotated databases. The following property holds:*

$$\mathbb{T}(\mathscr{D} \uplus \Delta\mathscr{D}) = \mathbb{T}(\mathscr{D}) \uplus \mathbb{T}(\Delta\mathscr{D})$$

Proof. Suppose $\mathscr{D}$ and $\Delta\mathscr{D}$ are:

$$\mathscr{D} = \{\!\{\langle t_1, \mathcal{P}_{t_1}\rangle, \ldots, \langle t_m, \mathcal{P}_{t_m}\rangle\}\!\}$$
$$\Delta\mathscr{D} = \mathbb{A}\,\mathscr{D} \cup \mathbb{\Delta}\,\mathscr{D} = \{\!\{\mathbb{\Delta}\,\langle t_{d_1}, \mathcal{P}_{d_1}\rangle, \ldots, \mathbb{\Delta}\,\langle t_{d_j}, \mathcal{P}_{d_j}\rangle\}\!\} \cup \{\!\{\mathbb{A}\,\langle t_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \mathbb{A}\,\langle t_{i_i}, \mathcal{P}_{i_i}\rangle\}\!\}$$

Then $\mathscr{D} \uplus \Delta\mathscr{D}$ is:

$$\begin{aligned}
&\mathscr{D} \uplus \Delta\mathscr{D}\\
=&\mathscr{D} - \mathbb{\Delta}\,\mathscr{D} \cup \mathbb{A}\,\mathscr{D}\\
=&\{\!\{\langle t, \mathcal{P}\rangle \mid \langle t, \mathcal{P}\rangle \in \mathscr{D}\}\!\} - \{\!\{\langle t, \mathcal{P}\rangle \mid \langle t, \mathcal{P}\rangle \in \mathbb{\Delta}\,\mathscr{D}\}\!\}\\
&\cup \{\!\{\langle t, \mathcal{P}\rangle \mid \langle t, \mathcal{P}\rangle \in \mathbb{A}\,\mathscr{D}\}\!\}\\
=&\{\!\{\langle t_1, \mathcal{P}_{t_1}\rangle, \ldots, \langle t_m, \mathcal{P}_{t_m}\rangle\}\!\}\\
&- \{\!\{\mathbb{\Delta}\,\langle t_{d_1}, \mathcal{P}_{d_1}\rangle, \ldots, \mathbb{\Delta}\,\langle t_{d_j}, \mathcal{P}_{d_j}\rangle\}\!\}\\
&\cup \{\!\{\mathbb{A}\,\langle t_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \mathbb{A}\,\langle t_{i_i}, \mathcal{P}_{i_i}\rangle\}\!\}
\end{aligned}$$

Then $\mathbb{T}(\mathscr{D} \uplus \Delta\mathscr{D})$:

$$\begin{aligned}
&\mathbb{T}(\mathscr{D} \uplus \Delta\mathscr{D})\\
=&\{\!\{t_1, \ldots, t_m\}\!\}\\
&- \{\!\{\mathbb{\Delta}\,t_{d_1}, \ldots, \mathbb{\Delta}\,t_{d_j}\}\!\}\\
&\cup \{\!\{\mathbb{A}\,t_{i_1}, \ldots, \mathbb{A}\,t_{i_i}\}\!\}
\end{aligned}$$

$\mathbb{T}(\mathscr{D}) \uplus \mathbb{T}(\Delta\mathscr{D})$ are:

$$\begin{aligned}
&\mathbb{T}(\mathscr{D}) \uplus \mathbb{T}(\Delta\mathscr{D})\\
=&\mathbb{T}(\mathscr{D}) \uplus \mathbb{T}(\mathbb{\Delta}\,\mathscr{D} \cup \mathbb{A}\,\mathscr{D})\\
=&\mathbb{T}(\mathscr{D}) - \mathbb{T}(\mathbb{\Delta}\,\mathscr{D}) \cup \mathbb{T}(\mathbb{A}\,\mathscr{D})\\
=&\{\!\{t_1, \ldots, t_m\}\!\}\\
&- \{\!\{\mathbb{\Delta}\,t_{d_1}, \ldots, \mathbb{\Delta}\,t_{d_j}\}\!\}\\
&\cup \{\!\{\mathbb{A}\,t_{i_1}, \ldots, \mathbb{A}\,t_{i_i}\}\!\}
\end{aligned}$$

Therefore, $\mathbb{T}(\mathscr{D} \uplus \Delta\mathscr{D}) = \mathbb{T}(\mathscr{D}) \uplus \mathbb{T}(\Delta\mathscr{D})$ □

Thus for the tuple correctness, the following property holds:

$$\mathbb{T}(Q(\mathscr{D}) \uplus \mathcal{I}(Q, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) = Q(\mathscr{D}') \qquad (\textbf{tuple correctness})$$

Recall $\mathcal{P}[Q, \mathcal{D}, D]$ defines an accurate provenance sketch $\mathcal{P}$ for Q wrt. to D, and ranges $\mathcal{D}$, and $D_{\mathcal{P}}$ is an instance of $\mathcal{P}$ which is the data covered by the sketch. We define a function $\mathbb{F}(\cdot)$ (**fragments extracting**)that takes as input a bag of annotated tuples and return all the sketches from this bag such that:

$$\mathbb{F}(\{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_y, \mathcal{P}_y\rangle\}\!\}) = \{\!\{\mathcal{P}_1, \ldots, \mathcal{P}_y\}\!\}$$

And the $\mathbb{F}(\cdot)$ function has the following property:

$$\mathbb{F}(\mathscr{D}_1 \uplus \mathscr{D}_2) = \mathbb{F}(\mathscr{D}_1) \cup \mathbb{F}(\mathscr{D}_2)$$

The proof of this property is similar to lemma A.3 where for $\mathbb{F}(\cdot)$, we focus on fragments instead of tuples.

We now define $\mathbb{S}(\cdot)$ as a function that takes as input a bag of fragments, where each fragment has a multiplicity at least 1, and return the set of fragments, which means that for each fragment in the input bag, the multiplicity of such fragment in the set is 1. For a query Q running over a annotated database $\mathscr{D}$, the annotated result is $Q(\mathscr{D})$. The fragments in the annotated result are $\mathbb{F}(Q(\mathscr{D}))$. Suppose the provenance sketch captured for this query given the ranges $\mathcal{D}$ is $\mathcal{P}[Q, \mathcal{D}, D]$, then we can get that:

$$\mathcal{P}[Q, \mathcal{D}, D] = \mathbb{S}(\mathbb{F}(Q(\mathscr{D})))$$

A provenance sketch covers relevant data of the database to answer a query such that:

$$Q(D) = Q(D_{\mathcal{P}[Q,\mathcal{D},D]}) = Q(D_{\mathbb{S}(\mathbb{F}(Q(\mathscr{D})))})$$

For a query running over a set of fragments, it is the same as running a bag of fragments where the ranges are exactly the same as

they appear in the fragment set but have different multiplicity. The reason is for a fragment appearing multiple times in the bag, when using this fragment to answer, they will be translate into the same expression in the `WHERE` clause multiple times concatenating with `OR`. And this expression appearing multiple times will be treated as a single one when the database engine evaluate the `WHERE`. For example, `WHERE (a BETWEEN 10 AND 20) OR (a BETWEEN 10 AND 20) OR (a BETWEEN 10 AND 20)` has the same effect as `WHERE (a BETWEEN 10 AND 20)` for a query. Thus, the following holds:

$$Q(D) = Q(D_{\mathbb{S}(\mathbb{F}(Q(\mathscr{D})))}) = Q(D_{\mathbb{F}(Q(\mathscr{D}))})$$

Thus, for the fragment correctness, we will prove the following property holds:

$$Q(D') = Q(D'_{\mathbb{S}(\mathbb{F}(Q(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q,\mathcal{D},\Delta\mathscr{D},\mathcal{S})))}) \quad \textbf{(fragment correctness)}$$

In the following part of this section, we will use inductive strategy to prove that the two properties will hold for every operator the incremental maintenance procedure maintain the sketches. The base case is a single table access operator. For other operators, we show the inductive steps by distinguishing the cases of tuple and fragment correctness for each operator.

A.1 Inductive Proof

A.1.1 Base Case. We start with Q^1, which is a single table R. We will show that for any $Q^1 \in \mathbb{Q}^1$, the **tuple correctness** and **fragment correctness** properties holds for table access operator. Suppose the relation and its annotated relation are:

$$R = \{\!\{t_1, \ldots, t_n\}\!\} \quad \mathscr{R} = \{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_n, \mathcal{P}_n\rangle\}\!\}$$

and delta relation and annotated delta relation are:

$$\triangle R = \{\!\{t_{i_1}, \ldots, t_{i_i}\}\!\} \quad \triangle R = \{\!\{t_{d_1}, \ldots, t_{d_j}\}\!\}$$

$$\triangle \mathscr{R} = \{\!\{\langle t_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \langle t_{i_i}, \mathcal{P}_{i_i}\rangle\}\!\} \quad \triangle \mathscr{R} = \{\!\{\langle t_{d_1}, \mathcal{P}_{d_1}\rangle, \ldots, \langle t_{d_j}, \mathcal{P}_{d_j}\rangle\}\!\}$$

Tuple Correctness .

$$\begin{aligned}
&\mathbb{T}(Q^1(\mathscr{R}) \uplus \mathcal{I}(Q^1, \mathcal{D}, \Delta\mathscr{R})) \\
=&\mathbb{T}(\mathscr{R} \uplus \Delta\mathscr{R}) \\
=&\mathbb{T}(\mathscr{R} - \triangle \mathscr{R} \cup \triangle \mathscr{R}) \\
=&\mathbb{T}(\mathscr{R}) - \mathbb{T}(\triangle \mathscr{R}) \cup \mathbb{T}(\triangle \mathscr{R}) \\
=&R - \triangle R \cup \triangle R \\
=&Q^1(R - \triangle R \cup \triangle R) \\
=&Q^1(R \uplus \Delta R) \\
=&Q^1(R')
\end{aligned}$$

Fragment Correctness . First, we will demonstrate the fragments after the incremental maintenance:

$$\begin{aligned}
&\mathbb{F}(Q^1(\mathscr{D})) \uplus \mathbb{F}(\mathcal{I}(Q^1, \mathcal{D}, \Delta\mathscr{D})) \\
=&\mathbb{F}(\mathscr{R}) \uplus \mathbb{F}(\mathcal{I}(Q^1, \mathcal{D}, \triangle \mathscr{D} \uplus \triangle \mathscr{D})) \\
=&\{\!\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}\!\} \uplus \triangle \{\!\{\mathcal{P}_{d_1}, \ldots, \mathcal{P}_{d_j}\}\!\} \uplus \triangle \{\!\{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_i}\}\!\}
\end{aligned}$$

Since for every sketch in $\{\!\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}\!\} \uplus \triangle \{\!\{\mathcal{P}_{d_1}, \ldots, \mathcal{P}_{d_j}\}\!\} \uplus \triangle \{\!\{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_i}\}\!\}$ associating a tuple, and the associated tuple will be inserted or deleted as the sketch does. Thus:

$$D'_{\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^1,\mathcal{D},\Delta\mathscr{D}))} = \{\!\{t_1, \ldots, t_n\}\!\} \uplus \triangle \{\!\{t_{i_1}, \ldots, t_{i_i}\}\!\} \uplus \triangle \{\!\{t_{d_1}, \ldots, t_{d_j}\}\!\}$$

Thus,

$$\begin{aligned}
&Q^1(D'_{\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))}) \\
=&\{\!\{t_1, \ldots, t_n\}\!\} \uplus \triangle \{\!\{t_{i_1}, \ldots, t_{i_i}\}\!\} \uplus \triangle \{\!\{t_{d_1}, \ldots, t_{d_j}\}\!\} \\
=&R \uplus \triangle R \uplus \triangle R
\end{aligned}$$

Since $Q^1(R') = Q^1(R \cup \Delta R) = R \uplus \triangle R \uplus \triangle R$, then we get for $Q^1 \in \mathbb{Q}^1$ that: $Q^1(D') = Q^1(D'_{\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^1,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))}) = Q^1(D'_{\mathbb{S}(\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^1,\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})$.

A.1.2 Inductive Steps. Assume the properties hold for $Q^i \in \mathbb{Q}^i$ such that the incremental maintenance procedure can correctly produce the tuples and provenance sketches.

For the **tuple correctness** ,

$$Q^i(D') = \mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))$$

For the **fragment correctness** ,

$$Q^i(D') = Q^i(D'_{\mathbb{S}(\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})$$

We will show that for $Q^{i+1} \in \mathbb{Q}^{i+1}$, where the $(i+1)$'s operator is on top of Q^i, both properties hold such that:

$$\begin{aligned}
Q^{i+1}(D') &= \mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
Q^{i+1}(D') &= Q^{i+1}(D'_{\mathbb{S}(\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})
\end{aligned}$$

A.1.3 Selection. Suppose the operator at level $i+1$ is an selection operator. Then we know that

$$Q^{i+1}(D) = \sigma_\theta(Q^i(D))$$

We assume that the $Q^{i+1}(D)$, $Q^{i+1}(\mathscr{D})$, $\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$ are as follows:

$$Q^{i+1}(D) = \{\!\{t_1, \ldots, t_n\}\!\}$$

$$Q^{i+1}(\mathscr{D}) = \{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_n, \mathcal{P}_n\rangle\}\!\}$$

$$\begin{aligned}
\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) =& \{\!\{\triangle \langle t_{d_1}, \mathcal{P}_{d_1}\rangle, \ldots, \triangle \langle t_{d_j}, \mathcal{P}_{d_j}\rangle\}\!\} \\
&\uplus \{\!\{\triangle \langle t_{i_1}, \mathcal{P}_{i_1}\rangle, \ldots, \triangle \langle t_{i_i}, \mathcal{P}_{i_i}\rangle\}\!\}
\end{aligned}$$

Tuple Correctness .

$$
\begin{aligned}
&Q^{i+1}(D')\\
=&\sigma_\theta(Q^i(D'))\\
=&\sigma_\theta\left(\mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\sigma_\theta\left(\mathbb{T}(Q^i(\mathscr{D})) \cup \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\sigma_\theta\left(\mathbb{T}(Q^i(\mathscr{D}))\right) \cup \sigma_\theta\left(\mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\mathbb{T}(\sigma_\theta(Q^i(\mathscr{D}))) \cup \mathbb{T}(\sigma_\theta(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \cup \mathbb{T}(\sigma_\theta(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \{\!\{t \mid t \in \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \wedge t \models \theta\}\!\}\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(\sigma_\theta(Q^i), \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}
$$

Fragment Correctness . We have the assumption that:

$$Q^i(D') = Q^i(D'_{\mathbb{F}(Q^1(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})$$

For $Q^{i+1}(D'_{\mathbb{S}(\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})$, we have the following:

$$
\begin{aligned}
&Q^{i+1}(D'_{\mathbb{S}(\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})\\
=&Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
=&Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D})\uplus\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})
\end{aligned}
$$

For $\langle t, \mathcal{P}\rangle \in Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$ every tuple associates a sketch. Then the tuples are the same from running query over the fragments covered by the sketch and over the database, such that:

$$
\begin{aligned}
&Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D})\uplus\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
&\qquad = \mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}
$$

Thus:

$$Q^{i+1}(D'_{\mathbb{S}(\mathbb{F}(Q^{i+1}(\mathscr{D})\uplus\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S})))}) = Q^{i+1}(D')$$

A.1.4 Projection. Suppose the operator at level $i+1$ is an projection operator. Then we have the following:

$$Q^{i+1}(D) = \Pi_A(Q^i(D))$$

We assume that the $Q^{i+1}(D)$, $Q^{i+1}(\mathscr{D})$, $\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$ are as follows:

$$Q^{i+1}(D) = \{\!\{t_1, \ldots, t_n\}\!\}$$

$$Q^{i+1}(\mathscr{D}) = \{\!\{\langle t_1, \mathcal{P}_1\rangle, \ldots, \langle t_n, \mathcal{P}_n\rangle\}\!\}$$

$$
\begin{aligned}
\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) =&\{\!\{\triangle\ \langle t_{d1}, \mathcal{P}_{d1}\rangle, \ldots, \triangle\ \langle t_{dj}, \mathcal{P}_{dj}\rangle\}\!\}\\
&\uplus \{\!\{\triangle\ \langle t_{i1}, \mathcal{P}_{i1}\rangle, \ldots, \triangle\ \langle t_{ik}, \mathcal{P}_{ik}\rangle\}\!\}
\end{aligned}
$$

Tuple Correctness .

$$
\begin{aligned}
&Q^{i+1}(D')\\
=&\Pi_A(Q^i(D'))\\
=&\Pi_A\left(\mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\Pi_A\left(\mathbb{T}(Q^i(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\Pi_A\left(\mathbb{T}(Q^i(\mathscr{D}))\right) \uplus \Pi_A\left(\mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\right)\\
=&\mathbb{T}(\Pi_A(Q^i(\mathscr{D})) \uplus \mathbb{T}(\Pi_A(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \{\!\{t \mid t' \in \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \wedge t'.A = t\}\!\}\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(\Pi_A(Q^i), \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}
$$

Fragment Correctness . For every $Q^i \in \mathbb{Q}^i$, we have the following:

$$Q^i(D') = Q^i(D'_{\mathbb{S}(Q^i(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})$$

From the projection operator maintenance rule, a projection operator on top of Q^i will not add or remove any fragment from $\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$. The projection operator will just output tuples associating sketches from $\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$ according to the expression list A. Thus:

$$
\begin{aligned}
&\mathbb{F}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{F}(\Pi_A(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{F}(\{\!\{\langle t, \mathcal{P}\rangle \mid \langle t', \mathcal{P}\rangle \in \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \wedge t'.A = t\}\!\})\\
=&\mathbb{F}(\mathcal{I}(\Pi_A(Q^{i+1}), \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
=&\mathbb{F}(\mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}
$$

Similarly:

$$
\begin{aligned}
&\mathbb{F}(Q^i(\mathscr{D}))\\
=&\mathbb{F}(\Pi_A(Q^i(\mathscr{D})))\\
=&\mathbb{F}(Q^{i+1}\mathscr{D})
\end{aligned}
$$

Then:

$$
\begin{aligned}
&\mathbb{F}(Q^i(\mathscr{D})) \uplus \mathbb{F}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\\
&\quad = \mathbb{F}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{F}((\mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
\Rightarrow&D'_{\mathbb{F}(Q^i(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))}\\
&\quad = D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))}\\
\Rightarrow&Q^i(D'_{\mathbb{F}(Q^i(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
&\quad = Q^i(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
\Rightarrow&Q^i(D') = Q^i(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
\Rightarrow&\Pi_A(Q^i(D')) = \Pi_A(Q^i(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))}))\\
\Rightarrow&Q^{i+1}(D') = Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})
\end{aligned}
$$

A.1.5 Cross Product. Suppose the operator at level $i+1$ is an cross product (join) operator, then we have the following:

$$Q^{i+1}(D) = Q_1^m(D) \times Q_2^{i-m}(D)$$

Tuple Correctness .

$$\begin{aligned}
&Q^{i+1}(D') \\
=&Q_1^m(D') \times Q_2^{i-m}(D') \\
=&\mathbb{T}(Q_1^m(\mathscr{D}) \uplus \mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
&\times \mathbb{T}(Q_2^{i-m}(\mathscr{D}) \uplus \mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
=&(\mathbb{T}(Q_1^m(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
&\times (\mathbb{T}(Q_2^{i-m}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
=&(\mathbb{T}(Q_1^m(\mathscr{D})) \times \mathbb{T}(Q_2^{i-m}(\mathscr{D}))) \\
&\uplus (\mathbb{T}(Q_1^m(\mathscr{D})) \times \mathbb{T}(\mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
&\uplus (\mathbb{T}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \times \mathbb{T}(Q_2^{i-m}(\mathscr{D})))) \\
&\uplus (\mathbb{T}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \times \mathbb{T}(\mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
=&\mathbb{T}(Q_1^m(\mathscr{D}) \times Q_2^{i-m}(\mathscr{D})) \\
&\uplus \mathbb{T}(Q_1^m(\mathscr{D}) \times \mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
&\uplus \mathbb{T}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \times Q_2^{i-m}(\mathscr{D})) \\
&\uplus \mathbb{T}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \times \mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
=&\mathbb{T}(Q_1^m(\mathscr{D}) \times Q_2^{i-m}(\mathscr{D})) \\
&\uplus \mathbb{T}\Big(Q_1^m(\mathscr{D}) \times \mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \\
&\quad \uplus \mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \times Q_2^{i-m}(\mathscr{D}) \\
&\quad \uplus \mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}) \times \mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})\Big) \\
=&\mathbb{T}(Q^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
=&\mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}$$

Fragment Correctness .

$$\begin{aligned}
&Q^{i+1}(D') \\
=&Q_1^m(D') \times Q_2^{i-m}(D') \\
=&Q_1^m(D'_{\mathcal{P}[Q_1^m, \mathcal{D}, D] \uplus \mathbb{F}(\mathcal{I}(Q_1^m, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))}) \\
&\times Q_2^{i-m}(D'_{\mathcal{P}[Q_2^{i-m}, \mathcal{D}, D] \uplus \mathbb{F}(\mathcal{I}(Q_2^{i-m}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))})
\end{aligned}$$

A.1.6 Aggregation. Suppose the operator at level $i+1$ is an aggregation function (any one of **sum**, **count**, **avg**, **min** and **max**), Then we have the following:

$$Q^{i+1}(D) = \gamma_{f(a);G}(Q^i(D))$$

We have the assumption that for $Q^i \in \mathbb{Q}^i$, the **tuple correctness** and **fragment correctness** hold. We will show that these properties still hold for $Q^{i+1} \in \mathbb{Q}^{i+1}$ when Q^{i+1} is an aggregation function.

To show the correctness of tuples and fragments, we focus on one group g. Let $Q_g^{i+1}(Q(D))$ to be an aggregation function works on $Q^i(D)$ and only focus on groups g, $t.G = g$. Thus, the two properties for g will be:

$$Q_g^{i+1}(D') = \mathbb{T}(Q_g^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q_g^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \quad \textbf{(tuple correctness)}$$

$$Q_g^{i+1}(D') = Q_g^{i+1}(D'_{\mathcal{P}[Q_g^{i+1}, \mathcal{D}, D] \uplus \mathbb{F}(\mathcal{I}(Q_g^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))}) \quad \textbf{(fragment correctness)}$$

Tuple Correctness . We start by showing for one group g, it exists before and update maintenance.

$$\begin{aligned}
&Q^i(D') = \mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
\Rightarrow&\gamma_{f(a);G}(Q^i(D')) = \mathbb{T}(\gamma_{f(a);G}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
\Rightarrow&\{\!\{(g \circ f(t)) \mid t = t'.a \wedge g \in G \wedge t'.G \in G \wedge t' \in Q^i(D')\}\!\} \\
&= \{\!\{(g \circ f(t)) \mid t = t'.a \wedge g \in G \wedge t'.G \in G \\
&\qquad \wedge t' \in \mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\}\!\}
\end{aligned}$$

Then, for one group g, we have:

$$\begin{aligned}
&\{\!\{(g \circ f(t)) \mid t = t'.a \wedge t'.G = g \wedge t' \in Q^i(D')\}\!\} \\
=&\{\!\{(g \circ f(t)) \mid t = t'.a \wedge t'.G = g \\
&\qquad \wedge t' \in \Big(\mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\Big)\}\!\} \\
=&\{\!\{(g \circ f(t)) \mid t = t'.a \wedge t'.G = g \\
&\qquad \wedge t' \in \Big(\mathbb{T}(Q^i(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\Big)\}\!\} \\
=&\{\!\{(g \circ f(t)) \mid \Big(t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(Q^i(\mathscr{D}))\Big) \\
&\qquad \uplus \Big(t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\Big)\}\!\} \\
=&\{\!\{(g \circ f(t)) \mid \Big(t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(Q^i(\mathscr{D}))\Big) \\
&\qquad \uplus \Big(t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))\Big)\}\!\} \\
=&\{\!\{(g \circ f(t)) \mid (t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(Q^i(\mathscr{D})))\}\!\} \\
&\qquad \uplus \{\!\{(g \circ f(t)) \mid (t = t'.a \wedge t'.G = g \wedge t' \in \mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\}\!\} \\
=&\gamma_{f(a);G=\{g\}}(\mathbb{T}(Q^i(\mathscr{D}))) \uplus \gamma_{f(a);G=\{g\}}(\mathbb{T}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
=&\mathbb{T}(\gamma_{f(a);G=\{g\}}(Q^i(\mathscr{D}))) \uplus \mathbb{T}(\gamma_{f(a);G=\{g\}}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))) \\
=&\mathbb{T}(Q_g^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(\gamma_{f(a);G=\{g\}}(Q^i), \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
=&\mathbb{T}(Q_g^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q_g^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}$$

Therefore, for one group g, the tuple correctness hold such that:

$$\begin{aligned}
&Q_g^{i+1}(D') \\
=&\mathbb{T}(Q_g^{i+1}(\mathscr{D})) \uplus \mathbb{T}(\mathcal{I}(Q_g^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})) \\
=&\mathbb{T}(Q_g^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q_g^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}$$

For each annotated tuple in $\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$, it only can belong to one group. There for every group $g \in G$, the property holds such that for aggregation function at level $i+1$:

$$Q^{i+1}(D') = \mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))$$

Fragment Correctness . For one group g, the incremental maintenance for aggregation $\gamma_{f(a);G=\{g\}}(Q^i(D))$ function will output two

annotated tuples:

$$\triangle\ \langle g \circ (f(a)), \mathcal{P}\rangle \qquad\qquad \triangle\ \langle g \circ (f(a)'), \mathcal{P}'\rangle$$

For $\triangle\ \langle g \circ (f(a)), \mathcal{P}\rangle$, it is equivalent to $\gamma_{f(a);G=\{g\}}(Q^i(\mathscr{D}))$, then $\mathbb{F}(\triangle\ \langle g \circ (f(a)), \mathcal{P}\rangle) = \mathbb{F}(\gamma_{f(a);G=\{g\}}(Q^i(\mathscr{D})))$. Thus,

$$\mathbb{F}(\triangle\ \langle g \circ (f(a)), \mathcal{P}\rangle) \uplus \mathbb{F}(\gamma_{f(a);G=\{g\}}(Q^i(\mathscr{D}))) = \emptyset$$

For $\triangle\ \langle g \circ (f(a)'), \mathcal{P}'\rangle$, the tuple is computed using $Q^i_g(\mathscr{D})$ and $\mathcal{I}(Q^i_g, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$. Thus, $\mathbb{S}(\mathbb{F}(\langle g \circ (f(a)'), \mathcal{P}'\rangle))$ contains all fragments from D' that compute $Q^{i+1}_g(D')$. Therefore, for group g:

$$\begin{aligned}
&Q^{i+1}_g(D')\\
=&Q^{i+1}_g(D'_{\mathbb{F}(\triangle\langle g\circ(f(a)'),\mathcal{P}'\rangle)})\\
=&Q^{i+1}_g(D'_{\emptyset\uplus\mathbb{F}(\triangle\langle g\circ(f(a)'),\mathcal{P}'\rangle)})\\
=&Q^{i+1}_g(D'_{\mathbb{F}(Q^{i+1}_g(\mathscr{D}))\uplus\mathbb{F}(\triangle\langle g\circ(f(a)),\mathcal{P}\rangle)\uplus\mathbb{F}(\triangle\langle g\circ(f(a)'),\mathcal{P}'\rangle)})\\
=&Q^{i+1}_g(D'_{\mathbb{F}(Q^{i+1}_g(\mathscr{D}))\uplus\mathbb{F}(\triangle\langle g\circ(f(a)),\mathcal{P}\rangle\uplus\triangle\langle g\circ(f(a)'),\mathcal{P}'\rangle)})\\
=&Q^{i+1}_g(D'_{\mathbb{F}(Q^{i+1}_g(\mathscr{D}))\uplus\mathbb{F}(\triangle\langle g\circ(f(a)),\mathcal{P}\rangle\uplus\triangle\langle g\circ(f(a)'),\mathcal{P}'\rangle)})\\
=&Q^{i+1}_g(D'_{\mathbb{F}(Q^{i+1}_g(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1}_g,\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})\\
=&Q^{i+1}_g(D'_{\mathbb{S}(\mathbb{F}(Q^{i+1}_g(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1}_g,\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})
\end{aligned}$$

A.1.7 Top-K. Suppose the operator at level $i+1$ is a top-k operator, we have

$$Q^{i+1}(D) = \tau_{k,o}(Q^i(D))$$

Tuple Correctness .

$$\begin{aligned}
&Q^{i+1}(D')\\
=&\tau_{k,o}(Q^i(D'))\\
=&\tau_{k,o}(\mathbb{T}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}) \uplus \mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D})) \uplus \tau_{k,o}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))
\end{aligned}$$

For $t^k \in \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))$, the k tuples are from $\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D})))$. Then

$$\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}))) \uplus \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))) = \emptyset$$

For $t \in \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}'))$, since the state ORDMAP$'$ contains all tuples corresponding to $\mathscr{D}'$, then, the $\mathbb{T}(\tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))$ contains all the k tuples for $\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}')))$. Therefore:

$$\begin{aligned}
&Q^{i+1}(D')\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D}'))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(D')))\\
=&\mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))\\
=&\emptyset \uplus \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}))) \uplus \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))) \uplus \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}))) \uplus \mathbb{T}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))) \uplus\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D}))) \uplus\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))) \uplus\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))\\
=&\mathbb{T}(\tau_{k,o}(Q^i(\mathscr{D})) \uplus \tau_{k,o}(\mathcal{I}(Q^i, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})))\\
=&\mathbb{T}(Q^{i+1}(\mathscr{D}) \uplus \mathcal{I}(Q^{i+1}, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S}))
\end{aligned}$$

Fragment Correctness .

For $\mathbb{F}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))$, the fragments are the same as from $\mathbb{F}(\tau_{k,o}(Q^i(\mathscr{D})))$, which is $\mathcal{P}[Q^{i+1}, \mathcal{D}, D]$. Then

$$\mathbb{F}(\tau_{k,o}(Q^i(\mathscr{D}))) \uplus \mathbb{F}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}))) = \emptyset$$

For $\mathbb{F}(\triangle\ \tau_{k,o}(\mathbb{E}(\text{ORDMAP}'))$, since the state ORDMAP$'$ contains all annotated tuples corresponding to $\mathscr{D}'$, then, the $\mathbb{S}(\mathbb{F}(\tau_{k,o}(\mathbb{E}(\text{ORDMAP}'))))$ contains all fragments of D' to get $\tau_{k,o}(Q^i(D'))$. Therefore:

$$\begin{aligned}
&Q^{i+1}(D')\\
=&Q^{i+1}(D'_{\mathbb{S}(\mathbb{F}(\tau_{k,o}(\mathbb{E}(\text{ORDMAP}'))))})\\
=&Q^{i+1}(D'_{\mathbb{F}(\tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))})\\
=&Q^{i+1}(D'_{\emptyset\uplus\mathbb{F}(\tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))})\\
=&Q^{i+1}(D'_{\mathbb{F}(\tau_{k,o}(Q^i(\mathscr{D})))\uplus\mathbb{F}(\triangle\tau_{k,o}(\mathbb{E}(\text{ORDMAP})))\uplus\mathbb{F}(\triangle\tau_{k,o}(\mathbb{E}(\text{ORDMAP}')))})\\
=&Q^{i+1}(D'_{\mathbb{F}(\tau_{k,o}(Q^i(\mathscr{D})))\uplus\mathbb{F}(\triangle\tau_{k,o}(\mathbb{E}(\text{ORDMAP}))\uplus\tau_{k,o}(\triangle\mathbb{E}(\text{ORDMAP}')))})\\
=&Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\tau_{k,o}(\mathcal{I}(Q^i,\mathcal{D},\Delta\mathscr{D},\mathcal{S})))})\\
=&Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}(\mathscr{D}))\uplus\mathbb{F}(\mathcal{I}(Q^{i+1},\mathcal{D},\Delta\mathscr{D},\mathcal{S}))})
\end{aligned}$$

B SQL-BASED STRATEGY

A pure SQL-based strategy is an option to update the provenance sketches incrementally. Accordingly, the state data required for operators is persisted as database tables. Then the incremental maintenance is modeled as running a series of queries over tables. For each operator of a query, queries are executed to process its input and to handle state data if required for both utilizing and updating the data.

This SQL-based approach has the advantages: first, there is no need to transfer data between the client and DBMS. What is more, DBMS can offer good plans and apply optimizations for query executions. Furthermore, data-heavy operations are executed inside the database, which reduces potential bottlenecks arising from data transfer across systems. All these advantages can enhance the efficiency of the incremental procedure.

However, this approach introduces challenges: first, it lacks flexibility to use specialized data structures to store operator state. All state data will be maintained in tables. But different operators' state

data can be kept in different data structures in-memory: all groups' average can be stored in a map and order and limit operators together can use binary search trees to fast access Top-K elements. In addition, state data storing in tables is less efficient compared to in in-memory data structures. For example, to update the average value of a group, the state data table should be scanned twice: accessing and updating data. While, these two operations can be done more faster if the state data is keep in a map in-memory. Moreover, incremental operations cannot always be efficiently expressed in SQL.

C APPENDIX I: QUERY LIST

C.1 Synthetic dataset query list

C.1.1 Different number of aggregation functions Q_{having}.

One aggregation function.

```
SELECT a, avg(b) AS ab
FROM r500
GROUP BY a
```

Two aggregation functions.

```
SELECT a, avg(b) AS ab
FROM r500
GROUP BY a
HAVING avg(c) < 1000
```

Three aggregation functions.

```
SELECT a, avg(b) AS ab
FROM r500
GROUP BY a
HAVING avg(c) < 1000 and avg(d) < 1200
```

Ten aggregation functions.

```
SELECT a, avg(b) AS ab
FROM r500
GROUP BY a
HAVING avg(c) < 1000 and avg(d) < 1200 and avg(e) > 0
    and avg(f) > 0 and avg(g) > 0 and avg(h) > 0
    and avg(i) > 0 and avg(j) > 0
```

C.1.2 Number of groups Q_{groups}.

50 groups.

```
SELECT a, avg(b) AS ab
FROM t1gb50g
GROUP BY a
HAVING avg(c) < 3
```

1K groups.

```
SELECT a, avg(b) AS ab
FROM t1gb1000g
GROUP BY a
HAVING avg(c) < 320
```

5K groups.

```
SELECT a, avg(b) AS ab
FROM t1gb5000g
GROUP BY a
HAVING avg(c) < 1600
```

500K groups.

```
SELECT a, avg(b) AS ab
FROM t1gb500000g
GROUP BY a
HAVING avg(c) < 1600
```

C.1.3 Join.

1-200K joins.

```
SELECT a, avg(b) AS ab
FROM (
    SELECT a AS a, b AS b, c AS c
    FROM t1gb50g
    WHERE b < 10
) tt JOIN tjoinhelp ON (a = ttid)
GROUP BY a
HAVING avg(c) < 10
```

1-2K joins.

```
SELECT a, avg(b) AS ab
FROM(
    SELECT a AS a, b AS b, c AS c
    FROM t1gbjoin WHERE b < 1000
) tt JOIN tjoinhelp ON (a = ttid)
GROUP BY a
HAVING avg(c) < 1000
```

1-20 joins.

```
SELECT a, avg(b) AS ab
FROM (
    SELECT a AS a, b AS b, c AS c
    FROM t1gb500000g
    WHERE b < 100000
) tt JOIN tjoinhelp ON (a = ttid)
GROUP BY a
HAVING avg(c) < 100000
```

C.1.4 Join selectivity: $Q_{joinsel}$.

1% selectivity.

```
SELECT a, avg(b) AS ab
FROM t1gbjoin JOIN tjoinhelppercent1 ON (a = ttid)
WHERE b < 1000
GROUP BY a
HAVING avg(c) < 1000
```

5% selectivity.

```
SELECT a, avg(b) AS ab
FROM t1gbjoin JOIN tjoinhelppercent5 ON (a = ttid)
WHERE b < 1000
GROUP BY a
HAVING avg(c) < 1000
```

10% selectivity.

```
SELECT a, avg(b) AS ab
FROM t1gbjoin JOIN tjoinhelppercent10 ON (a = ttid)
WHERE b < 1000
GROUP BY a
HAVING avg(c) < 1000
```

C.1.5 Fragment number: Q_{sketch}.

```
SELECT a, avg(b) as ab
FROM (
      SELECT a as a, b as b, c as c
      FROM t1gbjoin
      WHERE b < 1000) tt
JOIN tjoinhelp on (a = ttid)
GROUP BY a
HAVING avg(c) < 1000
```

C.1.6 Delta filter by selection push down Q_{selpd}.

```
SELECT a, avg(b) AS ab
FROM t1gb1000g
WHERE b < 1000
GROUP BY a
HAVING avg(c) < 300
```

C.1.7 End-to-end $Q_{endtoend}$.

```
SELECT a, avg(c) AS ac
FROM edb1
GROUP BY a
HAVING avg(c)> 1684845 AND avg(c) < 1686014;
```

C.2 Crimes dataset query list

Q1.

```
SELECT beat, year, count(id) AS crime_count
FROM crimes
GROUP BY beat, year
```

Q2.

```
SELECT district, community_area, ward, beat,
    count(beat) AS crime_count
FROM crimes
GROUP BY district, community_area, ward, beat
HAVING count(id) > 1000
```