In-memory Incremental Maintenance of Provenance Sketches

ABSTRACT

Provenance-based data skipping [27] 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.

1 INTRODUCTION

Database engines take advantage of physical design such as index structures, zone maps [22] 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 query's condition. However, as was demonstrated in [27], 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 for answering a query. In [27] 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 [27] that ensure that a sketch is *sufficient*, i.e., evaluating the query over the data represented by a 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 [27], 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 [27], the trade-offs for provenance sketches are different 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 [27], 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 databases shown Fig. 1 and query Q_{Top} that returns products whose total sale volume is greater than \$2000. Evaluating this query over the table shown in Fig. 1 produces a single result tuple* $(Apple, 5074)$. *The provenance of Q_{Top} in table* `sales` *are the two tuples for the Apple brand (tuples s_3 and s_4 shown with purple background), as the group for Apple is the only group that fulfills the having clause of the query. 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 partitions the table on attribute* `price` *based on ranges:*

$$\mathcal{R}_{price} = \{r_1 = [1, 600], r_2 = [601, 1000], \\ r_3 = [1000, 1500], r_4 = [1501, 20000]\}$$

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. $F_{\mathbf{sales},price}$ 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 the original 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 [27], 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

[1] In general, this is not the case for non-monotone queries. The safety check from [27] 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}.

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.

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)
```

While sketches have been demonstrated to be effective in improving query performance, 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. To be able to efficiently maintain the instrumented queries that generate sketches and, thus, maintain the sketch under updates, we develop incremental versions of relational algebra operators over sketch-annotated data.

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. Full maintenance is only beneficial if both queries and updates are executed in batches such that a maintained sketch can be used by a large enough number of queries before it becomes stale, thus, amortizing the cost of maintenance.

Note that the instrumented version of a query Q that creates a provenance sketch for Q are expressible in SQL. Consider for now a partition F of a table R accessed by Q and let $Q_{R,F}$ denote this query, generated using the rewrite rules from [27]. Theoretically it is possible to use existing view maintenance techniques 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 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**ncremetnal **M**aintenance of **P**rovenance Sketches). The input to this engine is a set of annotated delta tuples (tuples that were inserted / deleted) that we extract from a backend DBMS. The engine features incrementalized versions of operators with annotated semantics. To maintain the capture query $Q_{R,F}$ for a user query Q to update the sketch created by $Q_{R,F}$ at some point in the past, we extract the delta between the current version of the database and the database instance at the original time of capture (or the last time we maintained the sketch) and then feed this delta as input to our incremental engine to compute a delta for the sketch. We demonstrated that applying the sketch delta to the previous version of the sketch, yields a valid sketch for the current version of the database. To incrementally maintain the annotated result of an operator, our engine has to maintain state for each operator. Our implementation can persist such state in the database and load it into the incremental engine when needed. IMP also implements SQL implementations of the incremental maintenance rules and can outsource 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:

- We formalize incremental maintenance of data annotated with sketches and develop incremental versions of relational algebra operators for sketch-annotated data.

[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`.

- We implement these rules in IMP, an in-memory incremental engine for sketch maintenance build as an extension in GProM and develop several novel optimizations.
- We experimentally compare IMP against an SQL-based approach of full maintenance and against a baseline that does not use provenance sketches. 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 necessary background in Sec. 4 and discuss related work in Sec. 3. We formally define the maintenance problem studied in this work in Sec. 4.6 and formalize incremental maintenance for sketches in Sec. 4.5. In Sec. 5, we introduce incremental sketch maintenance rules for relational algebra operators and prove the correctness of these rules in Sec. 6. We discuss our in-memory implementation of IMP in Sec. 9, present experimental results in Sec. 10, and conclude in Sec. 13.

Boris says: We may not have the space for this here. Let's see whether we can discuss it somewhere else instead? When transactions are run under weaker isolation levels, e.g., using snapshot isolation as implemented by many DBMS including Postgres, then we need to ensure that a sketch used to run a query executed by a transaction will be based on the version of the database seen by this query. We present a solution for snapshot isolation that associates a sketch with a version of the database and maintains multiple versions of a sketch (each for a different version of the database). As sketches are small, this is reasonable. However, if the need arises, sketches can be garbage-collected using the same mechanism as used to delete old versions of tuples that are no longer visible to current transactions. If we retain sketches, then for DBMS that support time-travel this enables such sketches to be used to speed-up time-travel queries.

2 OVERVIEW OF IMP

Fig. 2 shows a overview of our IMP engine. At the beginning, users can input queries into the IMP with an option to indicate to capture, use or update the provenance sketches. Inside the middle ware, the queries will first be parsed to relational algebra trees, then the trees will go to one of three directions based on users' purpose. If provenance sketches need to be captured (blue pipeline), then rewriting queries into capture SQL and send this SQL to the backend and compute the provenance sketches. And if queries need to compute the results with sketches (green pipeline), SQL with sketches rewrite will be executed inside the use rewrite module, and the results are returned after executing SQL with sketches in the database. The last option is to update the provenance sketches, the queries will be forwarded to IMP engine (red pipeline) to update the sketches.

The maintenance procedure will be executed if a decision is made to update the provenance sketches. First, the IMP engine will fetch state data (State Data in the figure) stored in the database, load into main memory, and restore it into different data structures based on operators. Then, the engine will maintain sketches for each operator bottom-up starting from fetching delta (Delta Data in the figure) from database for each modified relation. For each operator, IMP will produce the output of delta sketches which will serve as the input of the next operator above. Finally, the delta provenance sketches (Delta PS in the figure) of the whole query will be generated. The updated provenance sketches can be computed based on the stale sketches and delta sketches. And the new version sketches will be kept in the database. During this procedure, state data of each operator will be maintained as well, and it will be materialized back into database.

3 RELATED WORK

Provenance. Provenance can be captured by annotating data and propagating the annotations using relational queries or by extending the database system [18] [29] [28]. Systems like GProM [3], Perm [13], Smoke [31], Smoked Duck [23], Links [11], ProvSQL[32] and DBNotes[4] can compute the provenance for relational queries. Provenance has been used to speed-up the queries in interactive visualization [31]. [27] present provenance-based data skipping (PBDS). The approach captures sketches over-approximating the provenance of a query and utilizes these sketches to speed-up subsequent execution of queries. We present the first approach for maintaining provenance sketches under updates, thus, enabling PBDS to be efficient for databases that are subject to updates.

Incremental View Maintenance(IVM). View maintenance has been studied extensively [5, 7, 12, 17, 20, 33]. [16] presented many techniques and applications of view maintenance. [34] surveyed maintenance algorithms and implementations. The initial work on view maintenance, e.g., [5, 7], used set semantics. This was later expanded to bag semantics (e.g., [9, 14]). We consider bag semantics in this work. Materialization has been studied for Datalog as well: [15] introduced the counting algorithms, [17] proposes the DRed (delete-rederive) algorithm, and [24] introduced the backward-forward algorithm. Incremental maintenance algorithms for iterative computations have been studied in [1, 6, 25, 26]. [19] proposed higher-order IVM [35] maintains aggregate views in temporal database. [30] proposes a general mechanism for aggregation functions. [2, 37] 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 (provenance sketch in our case) based on predication of future benefits resulting from using the artifact for speeding up operations. While a full self-tuning solution for sketches is beyond the scope of this work, we briefly discuss a basic cost model for sketch maintenance and its application to tuning sketches in Sec. 12.

Different strategies have been studied for maintaining views eagerly and lazily. For instance, [10] presented algorithms for deferred maintenance and [5, 8, 17] studied immediate view maintenance. Our approach supports both cases: if users provide an update statement, the system can immediately maintain sketches, or sketches can be updated lazily according to a delta between the current and a past database version when needed.

Maintaining Provenance. [36] presents a system for maintenance of provenance at in a distributed Datalog engine. In contrast to our work, [36] 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 enable new optimizations. Furthermore, as our goal of using provenance is improving query performance, we need to

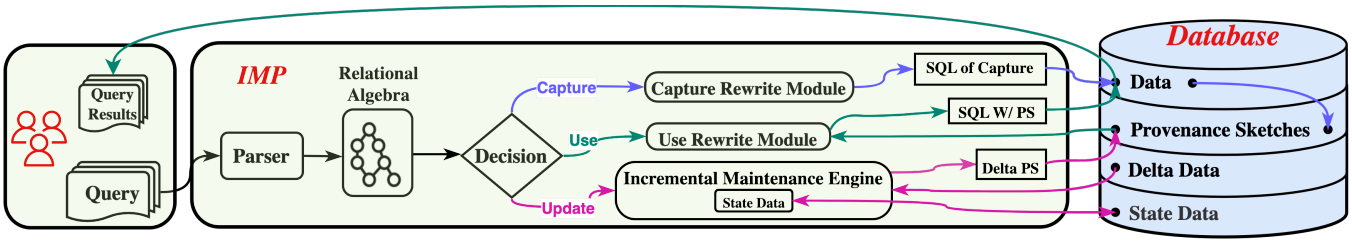


Figure 2: Overview of IMP: the system manages a set of sketch versions. For each incoming query, IMP determines whether to (i) capture a new sketches, (ii) use an existing non-stale sketch, or (iii) incrementally maintain a stale sketch and utilize the updated sketch to answer the query.

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.

4.1 The relational data model and algebra

A database schema $\mathbf{D} = (\mathbf{R_1}, \ldots, \mathbf{R_n})$ is a set of relation schemas $\mathbf{R}_1$ to $\mathbf{R}_n$. A relation schema $\mathbf{R}(a_1, \ldots, a_n)$ consists of a name ($\mathbf{R}$) 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 R is a function $\mathbb{U}^n \rightarrow \mathbb{N}$ mapping tuples to their multiplicity. The arity of $\mathbf{R}$, $arity(\mathbf{R})$, is the number of attributes in $\mathbf{R}$. We use $\{\!|\cdot|\!\}$ to denote bags and $t^n \in R$ to denote tuple t with multiplicity n in relation R. Fig. 3 shows the relational algebra in bag semantics 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 $e \rightarrow a$) 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. Union $R \cup S$ returns the bag union of tuples from relations R and S. Intersection $R \cap S$ returns the tuples which are both in relation R and S. Difference $R - S$ returns the tuples in relation R which are not in S. These set operations are only defined for inputs of the same arity. 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., $\gamma_{f(a)\rightarrow x,G}(R)$, renames $f(a)$ as x. Duplicate removal $\delta(R)$ removes duplicates. Top-K $\tau_{k,o}(R)$ returns the first k tuples from the relation R based on order by attribute list o. The position of a tuple among the order, we denote $\mathbf{pos}(t, R, o)$ to be a 0-based indexing function that return the first position of a tuple t, and o^t to denote the order of a tuple t. Thus, the function can be defined as $\mathbf{pos}(t, R, o) = \sum_{t' <_{o^t} t \wedge t^m \in R} m$.

Operator	Definition
σ	$\sigma_\theta(R) = \{\!\|t^n \| t^n \in R \wedge t \models \theta\|\!\}$
Π	$\Pi_A(R) = \{\!\|t^n \| n = \sum_{u.a=t} R(u)\|\!\}$
$\cup$	$R \cup S = \{\!\|t^{n+m} \| t^n \in R \wedge t^m \in S\|\!\}$
$\cap$	$R \cap S = \{\!\|t^{min(n,m)} \| t^n \in R \wedge t^m \in S\|\!\}$
$-$	$R - S = \{\!\|t^{max(n-m,0)} \| t^n \in R \wedge t^m \in S\|\!\}$
$\times$	$R \times S = \{\!\|(t \circ s)^{n*m} \| t^n \in R \wedge s^m \in S\|\!\}$
γ	$\gamma_{f(a),G}(R) = \{\!\|(t.G, f(G_t))^1 \| t \in R\|\!\}$ $G_t = \{(t_1.a)^n \| {t_1}^n \in R \wedge t_1.G = t.G\}$
δ	$\delta(R) = \{\!\|t^1 \| t \in R\|\!\}$
$\tau_{k,o}$	$\tau_{k,o}(R) = \{\!\|t^m \mid \mathbf{pos}(t, R, o) < k$ $\wedge m = min(R(t), k - \mathbf{pos}(t, R, o))\|\!\}$

Figure 3: Bag Relational Algebra

4.2 Range-based provenance sketch

We propose 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.

4.2.1 Range Partitioning. Given a set of intervals over the domains of a set of partitioning 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 this also covers the case where some relations are not partitioned by setting $\mathcal{R}_i = \{[min(\mathbb{D}(a_i)), max(\mathbb{D}(a_i))]\}$ (a single range covering all domain values).

4.2.2 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 $P(Q,D)$:

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

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 $\bigcup_{r\in\mathcal{P}} D_r$.*

Given a query Q over relation R, a provenance sketch $\mathcal{P}$ is a compact and declarative description of a superset of the provenance of a query (the instance $D_{\mathcal{P}}$ of $\mathcal{P}$). We call a sketch *accurate* if it only contains ranges whose fragments contain provenance. Note that we do not require that all relations of D are associated with a sketch. This is modeled in our framework by associating a relation R_i with a single range covering the whole domain of a_i as explained above. That is, associating each relation with a sketch is just for ease of presentation. As an example 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 instances $D_{\mathcal{P}}$, i.e., the data covered by the sketch, consists of all tuples contained in fragments f_3 and f_4 which are: $\{s_3, s_4, s_5\}$.

4.3 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 standard SQL updates, insert, and delete statements. A history $\mathcal{W}$ is a sequence of update operations:

$$\mathcal{W} = (U_1, \ldots, U_n)$$

For now we will consider linear histories produced by serializable concurrency control protocols and delay the discussion of how to handle weaker isolation levels to Sec. 11. Applied to an initial database state $D_0 = \emptyset$, a history defines a sequence of database states $\mathbf{D} = D_0, \ldots, D_n$ which we refer to as a database history. More formally, D_i for all $i > 0$ is defined as :

$$D_0 = \emptyset \wedge D_i = U_i(D_{i-1})$$

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

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

For a given delta ΔD, we use ΔD ($\mathbb{A}D$) to denote $\{\!\{\mathbb{A}t \mid \mathbb{A}t \in \Delta D\}\!\}$ ($\{\!\{\mathbb{A}t \mid \mathbb{A}t \in \Delta D\}\!\}$). Given a delta ΔD we can *apply* the delta to a database D denoted as $D \uplus \Delta D$:

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

Note that by definition of $\Delta(\cdot,\cdot)$ we have for any D_1 and D_2:

$$D_1 \uplus \Delta(D_1, D_2) = D_2$$

For a given database history $\mathbf{D}$, we will use ΔD_i to denote $\Delta(D_i, D_{i-1})$ and ΔD_{ij} to denote $\Delta(D_i, D_j)$.

4.4 Provenance Sketches Capture and Deltas

4.4.1 Capturing Provenance Sketches. Given a database D and a query Q and partitioning ranges $\mathcal{D}$ we use $C(Q,\mathcal{D},D)$ to denote running the capture query generated using the techniques from [27] to produce an **accurate provenance sketch** $\mathcal{P}$ for Q wrt. to D, ranges $\mathcal{D}$:

$$C(Q,\mathcal{D},D) = \mathcal{P}$$

4.4.2 Provenance Sketch Deltas. Analog to databases, we also define the delta for two provenance sketches $\mathcal{P}_1$ and $\mathcal{P}_2$ over the same range partitioning based on $\mathcal{R}$ on an attribute a of relation R:

$$\Delta(\mathcal{P}_1, \mathcal{P}_2) = \{\!\{\mathbb{A}\ r \mid r \in \mathcal{P}_2 - \mathcal{P}_1\}\!\} \cup \{\!\{\mathbb{A}\ r \mid r \in \mathcal{P}_1 - \mathcal{P}_2\}\!\}$$

For a given database history, we use $\mathcal{P}[Q,\mathcal{D}]_i$ to denote the version of the sketch for D_i. If the query and partition are clear from the context of irrelevant to the discussion, we will write $\mathcal{P}_i$ instead of $\mathcal{P}[Q,\mathcal{D}]_i$. Also we make use of $\uplus$ to denote application of deltas, $\Delta\mathcal{P}_i$ denote $\Delta(\mathcal{P}_{i-1}, \mathcal{P}_i)$, and $\Delta\mathcal{P}_{ij}$ to denote $\Delta(\mathcal{P}_i, \mathcal{P}_j)$.

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:

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

4.5 Sketch-Annotated Databases And Deltas

The incremental maintenance approach for sketches we present in this paper is build on relations whose tuples are annotated with provenance sketches. We then define an incremental semantics for maintaining the results of relational operators over such annotated relations and will demonstrate that this semantics correctly maintains sketches. Before discussing this semantics and formalizing the problem solved in this work, we now define such annotated relations as well as an operator $\mathbf{annotate}(R,\mathcal{R},a)$ that annotates each tuple in a regular input relation R with the fragment from a partitioning of attribute a on ranges $\mathcal{R}$ it belongs too and define deltas for annotated relations. This operator will be used to generate inputs

for incremental relational algebra operators operating on annotated relations.

DEFINITION 4.3 (SKETCH ANNOTATED RELATION). *A sketch annotated relation $\mathscr{R}$ 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 function $\mathbf{annotate}(R, \mathcal{R}, a)$ that annotates each tuple with the singleton set containing the range it's value in attribute a belongs to.

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 $\mathscr{R}$ 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 $\Delta\mathscr{R}_{ij}$ as:

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

Intuitively, $\Delta\mathscr{R}_{ij}$ contains all tuples from R that differ between D_i and D_j tagged with $\mathbb{\Delta}$ or Δ 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 $\mathbf{annotate}(D, \mathcal{D}) = \{\mathbf{annotate}(R, \mathcal{D}) \mid R \in D\}$.

EXAMPLE 4.2. *Continuing with Ex. 4.1, the annotated version of ΔD_2 is:*

$$\{\langle \mathbb{\Delta}\ s_8, \{r_3\}\rangle\}$$

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

4.6 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 $\Delta\mathscr{D}$ for the ranges $\mathcal{R}$ 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 $\Delta\mathscr{D}$, the maintenance procedure should return a delta for the sketch $\mathcal{P}$ and an updated state $\mathcal{S}'$ (that will be passed again to the procedure when it processes the next annotated delta).

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 $\Delta\mathscr{D}$, and returns an updated state $\mathcal{S}'$ and a provenance sketch delta $\Delta\mathcal{P}$:*

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

We require that for any Q, database D, delta ΔD, and provenance sketch $\mathcal{P}$ for Q and D, that the maintenance procedure has to produce $\Delta\mathcal{P}$ such that

$$C(Q, \mathcal{D}, D \cup \Delta D) \subseteq \mathcal{P} \uplus \Delta\mathcal{P}$$

We call a maintenance procedure **accurate** *iff $\mathcal{P} \uplus \Delta\mathcal{P} = C(Q, \mathcal{D}, D \cup \Delta D)$.*

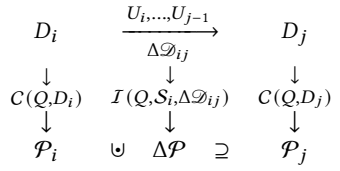


Figure 4: Incremental Maintenance V.S. Full Maintenance

Consider two database versions D_i and D_j. Fig. 4 shows shows how incremental and full maintenance relate to each other. Intuitively, a maintenance procedure has to produce a sketch delta $\Delta\mathcal{P}$ that if applied to the sketch $\mathcal{P}_i$ for the database before the updates represented by $\Delta\mathscr{D}$ returns a sketch $\mathcal{P} \uplus \Delta\mathcal{P}$ that over-approximates the sketch $\mathcal{P}_j$ produced by rerunning the capture operation over the updated database. As [27] demonstrated 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. As we will show later when discussing optimizations, over-approximations, while resulting is less benefits when the sketch is used to answer a query, can be produced at lower maintenance cost which enables trading maintenance cost for query performance.

5 INCREMENTAL ANNOTATED OPERATOR 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 $\mathcal{I}(O, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$ ($\mathcal{I}(Q, \mathcal{D}, \Delta\mathscr{D}, \mathcal{S})$) to denote the result of evaluating O (Q) over the annotated delta $\Delta\mathscr{D}$ using the state $\mathcal{S}$. We will drop $\mathcal{S}$ and $\mathcal{D}$ if they are clear from the context and for operators that do not store any state. Our incremental maintenance procedure evaluates a query expressed in relational algebra producing an updated state and outputting a delta to the query result where each delta row is annotated with a partial sketch delta. The procedure combines these partial sketch deltas to generate the final result $\Delta\mathcal{P}$. After introducing the semantics for each supported operator, we will prove that they form indeed an incremental maintenance procedure.

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 $\mathcal{S} : \mathcal{D} \rightarrow \mathbb{N}$ from the ranges used by the partitions used in the sketch of the query. used in sketches for the query to integers. This integers encode a counter for a fragment. Intuitively, this counter indicates the number of result tuples that have this fragment 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).

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

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:

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

Any newly inserted (deleted) tuple whose sketch includes r increases (decreases) the count for r. That is the total cardinality of such inserted tuples (bag $\mathbb{\Delta}\ \mathscr{D}$ and $\Delta\ \mathscr{D}$, respectively) has to be add (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 to any results and needs to be removed. Thus, $\Delta\mathcal{P}$ is computed as shown below:

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

Example 5.1. *Reconsider our running example data and partitioning based on $\mathcal{R}_{price}$ from Ex. 1.2. Assume that currently, there are two result tuples t_1 and t_2 of a query Q that have fragment r_2 (based on range $r_2 = [601, 1000]$ for the price attribute in the input) in their sketch and one result tuple t_3 that has r_1 and r_2 in its sketch. Then the current state sketch for the query is $\mathcal{P} = \{r_1, r_2\}$ and the state of μ would be:*

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

Assume that we are processing a delta $\mathbb{\Delta}\langle t_3, \{r_1, r_2\}\rangle$ (deleting tuple t_3 from the output of the query. The updated counts are:

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

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 Table Access Operator

The incremental version of the table access operator R returns the annotated delta $\Delta\mathscr{R}$ passed as part of $\Delta\mathscr{D}$ to the maintenance procedure for R unmodified. This operator does not maintain any state.

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

5.3 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 input delta tuple on a list of expressions A. Each tuple in the output depends on a single tuple in the input under bag semantics. Thus, the projection operator, for each annotated delta tuple $\Delta\langle t, \mathcal{P}\rangle$, we project t on A and just propagates $\mathcal{P}$ unmodified ($t.A$ in the result depends on the same subset of the data as t in the input of the projection):

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

5.4 Selection

Like projection, maintaining selection operator does not require state data as each output tuple depends on a single input tuple and, thus, each annotated delta tuple in the input can be dealt with independent of every other delta tuple in the input. The selection operator returns all input delta tuples that fulfill the selection condition unmodified and filters out all delta tuples that do not fulfill the selection condition.

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

5.5 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 state of Q_2 ($[\![Q_2]\!]_{\mathscr{D}}$, (ii) joining the delta of the Q_2 with the current state of Q_1 ($[\![Q_1]\!]_{\mathscr{D}}$), (iii) joining the deltas of Q_1 and Q_2. For the last case, for each pair of tuples from the deltas of Q_1 and Q_2 there are four possible cases depending on which of the two tuples 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 case (ii) and (iii). These rules have been discussed in [10, 14, 19, 21]. Finally, for $\mathbb{\Delta}\ s$ and $\mathbb{\Delta}\ t$ and the symmetric case $\Delta\ s$ and $\mathbb{\Delta}\ t$, no output delta has to be produced as

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

Pengyuan says: Can we write in the following way? it is clear to specify the delta insert and delta delte:

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

EXAMPLE 5.2. *Fig. 5 shows two annotated table $\mathscr{R}$ and $\mathscr{S}$. Tuples with − (red color) will be deleted from the relations and tuples with + (red color) will be inserted into the relations after update. Suppose a query is* `SELECT r,sum(s) as summation FROM R, S`. *Relations $\mathscr{R} \times \mathscr{S}$ and $\mathscr{R}' \times \mathscr{S}'$ show the annotated result of $\mathscr{R} \times \mathscr{S}$ before and after updated. When we incrementally compute the output for cross product operator, we should calculate $\Delta(\mathscr{R} \times \mathscr{S})$. Relation $\Delta(\mathscr{R} \times \mathscr{S})$ in Fig. 5 lists all annotated delta tuples of $R \times S$ as well as tuples' sketches. If we compute the $\mathscr{R} \times \mathscr{S} \uplus \Delta(\mathscr{R} \times \mathscr{S})$, the result contains the same annotated tuples as $\mathscr{R}' \times \mathscr{S}'$ does. The relation $\Delta(\mathscr{R} \times \mathscr{S})$ serves as the output of the cross product operator and the input of aggregation operator as well (which will be discussed in next example).*

5.6 Aggregation: sum, count, and average

For the aggregation operator, we have to store state about aggregation results for individual groups and the contribution of fragments from a provenance sketch towards an 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 from group-by attribute values 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 first **sum**, **count**, and **avg** that share essentially the same type of state data and then will discuss **min** and **max**. We will discuss the state for an aggregation with a single aggregation function. The state for multiple aggregation functions is analog.

5.6.1 **sum** *Function.* 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:

$$\mathcal{S}[g] = (sum\colon s, cnt\colon c, ps\colon \mathcal{P}, map\colon \mathscr{F}_g)$$

Here, *sum* and *cnt* store the current sum and count for the group, *ps* stores the groups sketch, and *map* stores $\mathscr{F}_g$, the map $\mathcal{D} \to \mathbb{N}$ introduced above that tracks for each range $r \in \mathcal{D}$ how many input tuples from $Q(D)$ contributing to the group have r in their sketch. We now first describe how to create the state for the aggregation operator based on a database instance. Once the state has been initialized, the operator can process annotated deltas.

State Data Initialization. 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

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

That is, s and c 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. Given the state of the aggregation operator initialized as explained above, 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 $\Delta\mathscr{D}$.

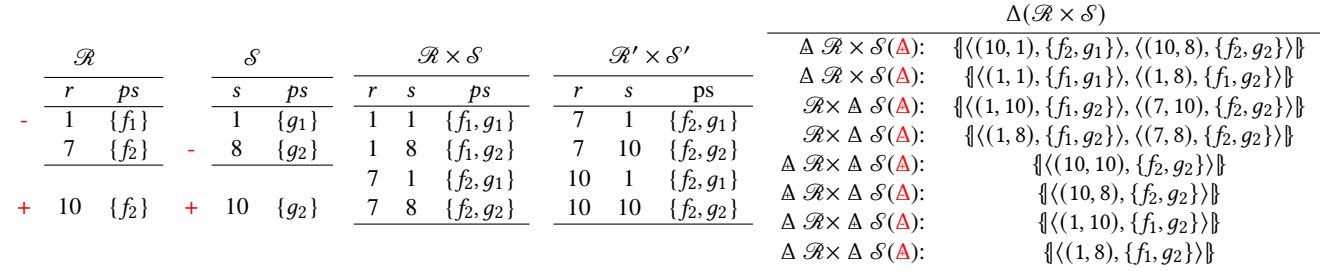


Figure 5: Incrementally maintain cross product

We use $\mathcal{G}_{\Delta Q}$ to denote the set of group-by values 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. Afterwards, we will discuss the two special cases when a new group is created or the last input tuple belonging to a group is deleted.

Updating an existing group. Assume that the current state for group g is:

$$\mathcal{S}[g] = (sum : s, cnt : c, ps : \mathcal{P}, map : \mathcal{F}_g)$$

The updated state produced by the operator is:

$$\mathcal{S}[g] = (sum : s', cnt : c', ps : \mathcal{P}', map : \mathcal{F}'_g)$$

The updated sum (cnt) are produced by adding $t.a \cdot n$ (n) for each inserted input tuple of the aggregation: $\mathbb{A}\langle t, \mathcal{P}\rangle^n \in \Delta Q_g$ and subtracting this amount for each deleted tuple: $\mathbb{A}\langle t, \mathcal{P}\rangle^n \in \Delta Q_g$:

$$c' = \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} n - \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} n$$
$$s' = \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} t.a \cdot n - \sum_{\mathbb{A}\langle t,\mathcal{P}\rangle^n \in \Delta Q_g} t.a \cdot n$$

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

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

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

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

Based on $\mathcal{S}'[g]$ we then output a pair of annotated delta tuples that deletes the previous aggregation result for the group and inserts the updated aggregation result:

$$\mathbb{A}\langle(g \circ (s)), \mathcal{P}\rangle \qquad \mathbb{A}\langle(g \circ (s')), \mathcal{P}'\rangle$$

Creating a new group. For groups g that are not in $\mathcal{S}$, we initialize the state for g as shown below:

$$\mathcal{S}[g] = (sum : 0, cnt : 0, ps : \emptyset, map : \emptyset)$$

and only output $\mathbb{A}\langle(g \circ (s')), \mathcal{P}'\rangle$

Deleting an existing group. An existing group gets deleted if $\mathcal{S}[g].cnt \neq 0$ and $\mathcal{S}'[g].cnt = 0$. In this case we only output $\mathbb{A}\langle(g \circ (s)), \mathcal{P}\rangle$.

Average and Count. For average we maintain the same state as for sum. The only difference is that the updated average is computed as $\frac{\mathcal{S}'[g].sum}{\mathcal{S}'[g].cnt}$. For count we only maintain the count and output $\mathcal{S}'[g].count$.

5.7 Aggregation: minimum and maximum

As **min** and **max** are symmetric, we only discuss **min** here. Consider an aggregation $\gamma_{\mathbf{min}(a),G}(Q)$. For insertions we simply have to check whether the inserted tuple's a value is smaller than the current minimum. If yes, then we update the minimum. Otherwise, no delta tuple is returned. To deal with deletions, if the tuple with the current minimum value is deleted we have to update the aggregation result to be the next larger value from the input (and update a group's sketch accordingly). For that we maintain a sorted map of the inputs of the aggregation for each group. The rationale for using a sorted map instead of a heap is that we need to be able to efficiently delete entries. We will discuss in optimizations for **min** and **max** that reduce the storage requirements at the cost of only being able to tolerate a certain number of deletions in Sec. 9.

State Data. As mentioned above, the state data for **min** needs to be able to for each group to (i) identify the current minimum and (ii) to update statistics for ranges $r \in \mathcal{D}$ and the multiplicity of tuples from the input of aggregation. We use a balanced binary search tree, but any other sorted map data structure with $O(logn)$ lookup, insertion, and deletion could be used as well. For each group we store the following data:

$$\mathcal{S}[g] = (orderedmap\colon om, ps\colon \mathcal{P}, map\colon \mathcal{F}_g)$$

Here $\mathcal{F}_g$ keeps track of the number of tuples for each range as described for **sum**.

Rule. Given state data $\mathcal{S}$, group identity values set $\mathcal{G}$, an input tuple $\Delta\langle t, \mathcal{P}\rangle$, and two output tuples $\mathbb{A}\langle t_1, \mathcal{P}_1\rangle$ and $\mathbb{A}\langle t_2, \mathcal{P}_2\rangle$ where $t = (g, a), t_1 = (g, m_1), t_2 = (g, m_2), g \in \mathcal{G}, m_1 = \mathcal{S}[g].orderedmap.min()$,

$m_3 = \mathcal{S}[g].orderedmap.nextMin()$[3], $\mathcal{P}_1 = \mathcal{S}[g].ps$, $\mathcal{F}_g = \mathcal{S}[g].map$, $\mathcal{F}_g{}' = upd(\mathcal{F}_g, \Delta\langle t, \mathcal{P}\rangle)$, and $\mathcal{P}_2 = \{f | \mathcal{F}_g{}' > 0\}$, the rule r_{min} is defined as follows:

$$\mathcal{I}(\gamma_{\mathbf{min}(a),G}(Q), \mathcal{S}, \Delta\langle t, \mathcal{P}\rangle) = \{\!\!\{\mathbb{\Delta}\langle t_1, \mathcal{P}_1\rangle, \mathbb{\Delta}\langle t_2, \mathcal{P}_2\rangle\}\!\!\} \qquad (r_{min})$$

$$where \begin{cases} m_2 = m_1, \ if\ \Delta = \mathbb{\Delta} \wedge m_1 < a \\ m_2 = a, \ if\ \Delta = \mathbb{\Delta} \wedge m_1 >= a \\ m_2 = m_1, \ if\ \Delta = \mathbb{\Delta} \wedge a > m_1 \\ m_2 = m_3, \ if\ \Delta = \mathbb{\Delta} \wedge a = c_1 \end{cases}$$

From the **min** rule, it is obvious that the input and output annotated tuples should be in the same group. The outputs should also contain two tuples: $\mathbb{\Delta}\langle t_1, \mathcal{P}_1\rangle$ and $\mathbb{\Delta}\langle t_2, \mathcal{P}_2\rangle$. Based on the attribute the **min** function works on, if the value of input instance is greater than the current min c_1, no matter the input is an insertion or deletion, m_1 and m_2 in outputs will be exactly the same, because the input cannot be the minimal among all values of its group. If the value of input is less than the minimal value, clearly, this input contains the minimal value after maintenance. Thus in this case, m_2 will be unknown at this point. If the input is a insertion, then it will be in this group and actually will be the minimal value after maintenance. While if the input is deleted from the group, it requires to know the next minimal value to be the minimal of this group after removing the current minimal value. The next minimal value can be obtained from the ordered map by two operation $\mathcal{S}[g].orderedmap.delete()$ and $\mathcal{S}[g].orderedmap.min()$. Two special cases of **sum** exists for **min** aggregation, and we apply the same strategy to deal with them: if the input tuple is the first one in the group, the output contains only an inserted annotated tuple, and only one deleted tuple in the output if the last tuple of a group is deleted.

Update State Data. For each tuple inserted, the aggregate attribute value is inserted into the tree, and delete the input value if it is an deletion. And for first tuple in the group, an new map entry is created and drop the group entry if the last tuple is deleted from the group.

max function. For **max**, it is symmetric to **min**, instead each time we compare the input value with min, we find the max of a group and the potential next max value if the input is a deletion.

EXAMPLE 5.3. *Continued with Ex. 5.2, when it comes to the aggregation operator, the state data before maintain the aggregation is shown as $\mathcal{S}$ in Fig. 6, which is built from $\mathcal{R} \times \mathcal{S}$. The input to maintain the aggregation are annotated tuples from $\Delta(\mathcal{R} \times \mathcal{S})$ (shown in Fig. 5). We process the input by first insertion (marked as $\mathbb{\Delta}$ in red color) then followed by deletion(marked as Δ in red color). During incrementally update aggregation operator, a new group with identity* 10 *will be added to the state data, because all tuples related to group identity g = 10 are inserted, and the summation of s is 11. Group identity of g = 1 will be removed from the state data. For group with identity g = 1, there are tuples of both insertion and deletion from the input: $\mathbb{\Delta}$((1, 10), (1, 8)) and Δ((1, 1), (1, 8), (1, 10), (1,8)). After processing all these tuple for g = 1, there is no more group record in the state data. The group of identity g = 7 will be updated. Thus, in the output, there is an annotated tuple $\langle(10, 11), f_2, g_1, g_2\rangle$ indicates it is a new group and an deleted tuple $\langle(1, 9), , \rangle f_1, g_1, g_2$ and there are two annotated tuples for group g = 7 updating the summation of s from* 9 *to* 11.

[3]nextMin() will return the second minimum value from the ordered map, this will be used for deletion only if current minimal value will not exist in the map after computation.

5.8 Top-K

Top-K operator returns the first k tuples from ordered tuples. To maintain this operator, the strategy is simple: first we output current top k tuples as the deleted delta tuples in output bags, then we porcess the input delta tuples, and at last, new first k tuples are propagated to the output bag as inserted output tuples.

State Data. We use an ordered map to store all tuples based on order by attributes. Each distinct order by attribute list o servers as the key in the map, and value is a regular map in which all attributes plus the provenance sketch of this tuple will be the key, and value is the multiplicity of this tuple. It is easy to understand that the key of the ordered may: different values will be different keys. The value of sorted map is also a map. The reason is that all values of different tuples may be the same, while the annotated tuples built from tuple plus provenance sketch can be different. Thus, the key in the inner map is an annotated tuple. Since Top-K operator needs to know the order of tuples, we keep a count as the value for inner regular map to track number of tuples in the first K when propagate the result. For each distinct key in the ordered map, we store the follow data:

$$\mathcal{S} = (orderedvalue\colon o, map\colon m)$$

And each value in the ordered map is a map, for this map, we store the following data:

$$m = (annotTup : \langle t, \mathcal{P}\rangle, multiplicity : c)$$

Rule.

$$\mathcal{I}(\tau_{k,o}(Q), \Delta\mathcal{D}) = \{\!\!\{\mathbb{\Delta}\langle t, \mathcal{P}\rangle^k, \mathbb{\Delta}\langle t', \mathcal{P}'\rangle^k\}\!\!\}$$
$$t \in [\![\tau_{k,o}(Q)]\!]_D$$
$$\mathcal{P} \in C(\tau_{k,o}(Q), D, \mathcal{D})$$
$$\mathbb{\Delta}\langle t', \mathcal{P}'\rangle^k \in \mathcal{I}(Q, \Delta\mathcal{D})$$

Update State Data. Updating the state data occurs during the incremental processing when dealing all input annotated tuples. For an annotated tuple **t** with the order-by attribute value list o, we have the following rule to update the data:

$$\begin{cases} \mathcal{S}[o][\Delta\langle t, \mathcal{P}\rangle] += 1 \ \ if \Delta = \mathbb{\Delta} \\ \mathcal{S}[o][\Delta\langle t, \mathcal{P}\rangle] -= 1 \ \ if \Delta = \mathbb{\Delta} \end{cases}$$

Inserting a new entry. For a inserted annotated tuple, if the order-by value list is not in sorted map as $\mathcal{S}$, we insert an new entry into the sorted map with key being this order-by value list and value of the inner map initialized with one entry: annotated tuple as key and multiplicity being 1.

Deleting a existing entry. If we delete a tuple, inside the inner map, the annotated tuple is the only key and its multiplicity is 1, after deleting this tuple, the inner map will be empty. There is no entry for this order-by value list in the sorted map any more after deleting. Thus, the entry of the sorted map will be removed as well.

$\mathcal{S}$

$\mathcal{S}[1] = (sum: 9, cnt: 2, ps: \{f_1, g_1, g_2\}, map: \{f_1 \mapsto 2, g_1 \mapsto 1, g_2 \mapsto 1\})$
$\mathcal{S}[7] = (sum: 9, cnt: 2, ps: \{f_2, g_1, g_2\}, map: \{f_2 \mapsto 2, g_1 \mapsto 1, g_2 \mapsto 1\})$

$\mathcal{S}'$

$\mathcal{S}[7] = (sum: 11, cnt: 2, ps: \{f_2, g_1, g_2\}, map: \{f_2 \mapsto 2, g_1 \mapsto 1, g_2 \mapsto 1\})$
$\mathcal{S}[10] = (sum: 11, cnt: 2, ps: \{f_2, g_1, g_2\}, map: \{f_1 \mapsto 2, g_1 \mapsto 1, g_2 \mapsto 1\})$

$outputs$

$\Delta: \{\!\{\langle(7, 11), \{f_2, g_1, g_2\}\rangle, \langle(10, 11), \{f_2, g_1, g_2\}\rangle\}\!\}$
$\Delta: \{\!\{\langle(1, 9), \{f_1, g_1, g_2\}\rangle, \langle(7, 9), \{f_2, g_1, g_2\}\rangle\}\!\}$

Figure 6: Incrementally maintain aggregation

5.9 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.

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

6 CORRECTNESS PROOF

We denote Q^n to a query having operators of size n. Then Q^1 is at the maximal depth which is a table access operator. We denote $\mathcal{P}^i$ to be the provenance sketch for Q^i and $\mathcal{F}^i$ is the fragment-count mapping before incremental maintenance. Let $\Delta\mathcal{P}^i$ be the delta provenance sketch for Q^i, $\mathcal{P}^{i'}$ be the provenance sketch after maintenance. We define $\mathbb{F}(Q^i, D, t)$ to donate a accurate set of fragments where tuple t can be obtained by running the query Q^i over the fragments of database D such that $t \in Q^i(D_{\mathbb{F}(Q^i,D,t)})$.

For correctness of each operator rule, we demonstrate two aspects: first is that the incremental procedure always compute the correct delta fragments from its input and can calculate the latest provenance sketches for the delta database such that $\forall t \in Q^{i+1}: \mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^i, D', t)$. We call this fragments correctness. Second is that the tuples of the fragments of new provenance sektches can result in correct queries answers such that $\forall t: t \in Q^i(D') \Leftrightarrow t \in Q^i(D'_{\mathbb{F}(Q^i,D',t)})$. We name the second criteria tuples correctness. For these criteria, they illustrate that the query results are equivalent from both the updated database and the fragments of new database.

THEOREM 6.1. *Consider a delta annotated relation $\Delta\mathcal{D}$, the output of a operator, including both a bag of inserted tuples $\{\!\{\langle \Delta t, \mathcal{P}\rangle\}\!\}$ and a bag of deleted tuples $\{\!\{\langle \Delta t, \mathcal{P}\rangle\}\!\}$ and a fragment-count mapping $\mathcal{F}$ built for the provenance sketch of this operator over database before modified. Then after computing the provenance sketch of tuples in $\Delta\mathcal{D}$, each fragment's count is no less than 0 at any point if $\{\!\{\langle \Delta t, \mathcal{P}\rangle\}\!\}$ is calculated first then followed by $\{\!\{\langle \Delta t, \mathcal{P}\rangle\}\!\}$ such that:*

$$\forall f \in \mathcal{F}: \mathcal{F}[f] \geq 0$$

6.1 Fragments Correctness

[27] shows the correctness of tuples inside fragments of operators at size Q^{i+1} can be derived from operators Q^i for a given dababase D without updates such that:

$$\forall t \in Q^{i+1}: \mathbb{F}(Q^{i+1}, D, t) = \mathbb{F}(Q^i, D, t)$$

In this part, we show that this property holds for database under updates such that:

$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D', t) = \mathbb{F}(Q^i, D', t)$$

When $n = 1$, it is a table access operator.

$$\forall t \in Q^1: \mathbb{F}(Q^1, D', t) = \{\!\{t \mid t \in D'\}\!\}$$

Suppose for $n = i$, the property holds such that:

$$\forall t \in Q^i(D'): \mathbb{F}(Q^i, D', t) = \mathbb{F}(Q^{i-1}, D, t)$$

We will show for $n = i + 1$, it still holds beased on different operators of Q^{i+1}. We assume the delta database is ΔD and updated database is D' such that $D \uplus \Delta D = D'$

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

$$\mathcal{I}(Q^{i+1}, \Delta\mathcal{D}) = \{\!\{\Delta\langle t.A, \mathcal{P}\rangle\}\!\} \implies \Delta\mathcal{P}^{i+1} \subseteq \{f \mid f \in \mathcal{P}\}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^{i+1}, D, t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^i, D', t)$$

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

$$\mathcal{I}(Q^{i+1}, \Delta\mathcal{D}) = \begin{cases} \{\!\{\Delta\langle t, \mathcal{P}\rangle\}\!\}, & \text{if } t \models \theta \\ \emptyset, & \text{otherwise} \end{cases}$$
$$\implies \Delta\mathcal{P}^{i+1} \subseteq \begin{cases} \{f \mid f \in \mathcal{P}\}, & \text{if } t \models \theta \\ \emptyset, & \text{otherwise} \end{cases}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D', t) = \mathbb{F}(Q^i, D', t)$$

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

$$\mathcal{I}(Q^{i+1}, \Delta\mathcal{D}) = \{\!\{\Delta\langle s \circ t, \mathcal{P}_1 \uplus \mathcal{P}_2\rangle^{n \cdot m}\}\!\}$$
$$\implies \Delta\mathcal{P}^{i+1} \subseteq \{f_1 \uplus f_2 \mid f_1 \in \mathcal{P}_1 \wedge f_2 \in \mathcal{P}_2\}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D, t) \uplus \mathbb{F}(Q_2^{i-m}, D, t) \cup \Delta\mathcal{P}^{i+1}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D, t) \uplus \{f_1 \mid f_1 \in \mathcal{P}_1\}$$
$$\uplus\, \mathbb{F}(Q_2^{i-m}, D, t) \uplus \{f_2 \mid f_2 \in \mathcal{P}_2\}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D', t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D', t) \uplus \mathbb{F}(Q_2^{i-m}, D', t)$$

Aggregation Functions. $Q^{i+1}(D) =_G \gamma_{f(a)\to x}(Q^i(D))$

$$\mathcal{I}(Q^{i+1}, \mathcal{S}, \Delta\langle t, \mathcal{P}\rangle) = \{\!\{\Delta\langle t_1, \mathcal{P}_1\rangle, \Delta\langle t_2, \mathcal{P}_2\rangle\}\!\}$$
$$\implies \Delta\mathcal{P}^{i+1} = \{f_1 \mid f_1 \in \mathcal{P}_1\} \uplus \{f_2 \mid f_2 \in \mathcal{P}_2\}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q^i, D, t) \uplus \Delta\mathcal{P}^{i+1}$$
$$\forall t \in Q^{i+1}: \mathbb{F}(Q^i, D', t) = \mathbb{F}(Q^i, D', t)$$

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

$$\mathcal{I}(Q^{i+1}, \Delta\mathscr{D}) = \mathcal{I}(Q_1^m, \Delta\mathscr{D}) \cup \mathcal{I}(Q_2^{i-m}, \Delta\mathscr{D}) = \{\!\{\Delta\langle t_1, \mathcal{P}_1\rangle\}\!\} \cup \{\!\{\Delta\langle t_2, \mathcal{P}_2\rangle\}\!\}$$
$$\implies \Delta\mathcal{P}^{i+1} = \{f_1 \mid f_1 \in \mathcal{P}_1\} \uplus \{f_2 \mid f_2 \in \mathcal{P}_2\}$$
$$\forall t \in Q^{i+1} : \mathbb{F}(Q^i, D', t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D, t) \uplus \mathbb{F}(Q_2^{i-m}, D, t) \uplus \Delta\mathcal{P}^{i+1}$$
$$\forall t \in Q^{i+1} : \mathbb{F}(Q^i, D', t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D, t) \uplus \{f_1 \mid f_1 \in \mathcal{P}_1\}$$
$$\uplus\, \mathbb{F}(Q_2^{i-m}, D, t) \uplus \{f_2 \mid f_2 \in \mathcal{P}_2\}$$
$$\forall t \in Q^{i+1} : \mathbb{F}(Q^i, D', t) \uplus \Delta\mathcal{P}^{i+1} = \mathbb{F}(Q_1^m, D', t) \uplus \mathbb{F}(Q_2^{i-m}, D', t)$$

6.2 Tuples Correctness

[27] shows that the results of running queries over the database is equivalent to those of running queries over the fragments in the provenance sketch such that:

$$\forall t : t \in Q^i(D) = t \in Q^i(\mathbb{F}(Q^i, D, t))$$

In this part, we show that this property holds for database under updates such that:

$$\forall t : t \in Q^{i+1}(D') = t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)})$$

When $n = 1$, it is a table access operator.

$$\mathbb{F}(Q^1, D', t) = \mathbb{F}(Q^1, D, t) \uplus \{f \mid t \in \Delta D \wedge t.a \in f\}$$
$$Q^1(D'_{\mathbb{F}(Q^1, D', t)}) = Q^1(D'_{\mathbb{F}(Q^1, D, t) \uplus \{f \mid t \in \Delta D \wedge t.a \in f\}})$$
$$= Q^1(D'_{\mathbb{F}(Q^1, D, t)}) \uplus Q^1(D'_{\{f \mid t \in \Delta D \wedge t.a \in f\}})$$
$$= Q^1(D) \uplus Q^1(\Delta D) = Q^1(D \uplus \Delta D) = Q^1(D)$$

Suppose for $n = i$, the property holds such that:

$$\forall t : t \in Q^i(D') = Q^1(D'_{\mathbb{F}(Q^i, D', t)})$$

We will show for $n = i + 1$, it still holds beased on different operators of Q^{i+1}. We assume the delta database is ΔD and updated database is D' such that $D \uplus \Delta D = D'$

6.2.1 *Projection Operator.* $Q^{i+1}(D) \coloneqq \Pi_A(Q^i(D))$

$$\forall t : \; : \mathbb{F}(Q^{i+1}, db', tup) = \mathbb{F}(Q^i, D', t)$$

STOPHERE

$$Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)}) = Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D, t) \uplus \{f \mid f \in \mathcal{P} \wedge \mathcal{P} \in \mathcal{I}(Q^{i+1}, \Delta\mathscr{D})\}})$$
$$= Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D, t)}) \uplus Q^{i+1}(D'_{\{f \mid f \in \mathcal{P} \wedge \mathcal{P} \in \mathcal{I}(Q^{i+1}, \Delta\mathscr{D})\}})$$
$$= Q^{i+1}(D_{\mathbb{F}(Q^{i+1}, D, t)}) \uplus Q^{i+1}(\Delta D_{\mathbb{F}(Q^{i+1}, \Delta D, t)})$$
$$= Q^{i+1}(D_{\mathbb{F}(Q^i, D, t)}) \uplus Q^{i+1}(D'_{\{f \mid f \in \mathcal{P} \wedge \mathcal{P} \in \mathcal{I}(Q^{i+1}, \Delta\mathscr{D})\}})$$
$$= \Pi_A(D_{\mathbb{F}(Q^i, D, t)}) \uplus Q^{i+1}(D'_{\{f \mid f \in \mathcal{P} \wedge \mathcal{P} \in \mathcal{I}(Q^{i+1}, \Delta\mathscr{D})\}})$$
$$= \Pi_A(Q^i(D)) \uplus Q^{i+1}(D'_{\{f \mid f \in \mathcal{P} \wedge \mathcal{P} \in \mathcal{I}(Q^{i+1}, \Delta\mathscr{D})\}})$$

PROOF.

$$Q^{i+1}(D') = \Pi_A(Q^i(D'))$$
$$\Leftrightarrow \forall t, t' : t \in Q^{i+1}(D') \wedge t' \in (Q^i(D')) \wedge t = t'.A$$
$$\forall t : t \in Q^{i+1}(D') \Leftrightarrow t' \in Q^i(D') \wedge t = t'.A$$
$$\overset{\text{rule } r_{prj}}{\implies} \forall t : \mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^i, D', t') \wedge t = t'.A \qquad (T_{pro})$$

$$\forall t : t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)})$$
$$\Leftrightarrow \forall t : t \in \Pi_A(Q^i(D'_{\mathbb{F}(Q^{i+1}, D', t)}))$$
$$\Leftrightarrow \forall t : t' \in Q^i(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \wedge t = t'.A$$
$$\overset{T_{pro}}{\Leftrightarrow} \forall t : t' \in Q^i(D'_{\mathbb{F}(Q^i, D', t')}) \wedge t = t'.a \wedge t = t'.A$$
$$\Leftrightarrow \forall t : t' \in Q^i(D'_{\mathbb{F}(Q^i, D', t')}) \wedge t = t'.A$$
$$\overset{HP}{\Leftrightarrow} \forall t : t' \in Q^i(D') \wedge t = t'.A$$
$$\Leftrightarrow \forall t : t \in Q^{i+1}(D')$$

□

6.2.2 *Selection Operator.* $Q^{i+1}(D) \coloneqq \sigma_\theta(Q^i(D))$

PROOF.

$$\forall t : t \in Q^{i+1}(D') \Leftrightarrow t \in \sigma_\theta(Q^i(D')) \Leftrightarrow t \in Q^i(D') \wedge t \models \theta$$
$$\overset{\text{rule } r_{sel}}{\implies} \forall t : \mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^i, D', t) \qquad (T_{sel})$$

$$\forall t : t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)})$$
$$\overset{T_{sel}}{\Leftrightarrow} \forall t : t \in Q^{i+1}(D'_{\mathbb{F}(Q^i, D', t)})$$
$$\Leftrightarrow \forall t : t \in \sigma_\theta(Q^i(D'_{\mathbb{F}(Q^i, D', t)}))$$
$$\Leftrightarrow \forall t : t \in Q^i(D'_{\mathbb{F}(Q^i, D', t)}) \wedge t \models \theta$$
$$\overset{HP}{\Leftrightarrow} \forall t : t \in Q^i(D') \wedge t \models \theta$$
$$\Leftrightarrow \forall t : t \in \sigma_\theta(Q^i(D'))$$
$$\Leftrightarrow \forall t : t \in Q^{i+1}(D')$$

□

6.2.3 *Cross Product Operator.* $Q^{i+1}(D) = Q_1^m(D) \times Q_2^{i-m}D \wedge m \leq i$

For cross product operator, it is assumed that a relation will use the same attribute, and same ranges to partition the relation if it is accessed multiple times.

PROOF.

$$
\begin{aligned}
&\forall t: t \in Q^{i+1}(D') \\
\Leftrightarrow &\forall t: t \in Q_1^m(D') \times Q_2^{i-m}(D') \\
\Leftrightarrow &\forall t, t_1, t_2: t \in Q^{i+1}(D') \wedge t_1 \in Q_1^m(D') \wedge t_2 \in Q_2^{i-m}(D') \\
&\quad \wedge t = t_1 \circ t_2 \\
\overset{r_{cro}}{\Longrightarrow} &\mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^m, D', t_1) \cup \mathbb{F}(Q^{i-m}, D', t_2) \qquad (T_{cro})
\end{aligned}
$$

$$
\begin{aligned}
&\forall t: t \in Q^{i+1}(D') \\
\Leftrightarrow &\forall t: t \in Q_1^m(D') \times Q_2^{i-m}(D') \\
\Leftrightarrow &\forall t, t_1, t_2: t_1 \in Q_1^m(D') \wedge t_2 \in Q_2^{i-m}(D') \wedge t = t_1 \circ t_2 \\
\overset{HP}{\Leftrightarrow} &\forall t, t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q_1^m, D', t_1)}) \wedge t_2 \in Q_2^{i-m}(D'_{\mathbb{F}(Q_2^{i-m}, D', t_2)}) \\
&\quad \wedge t = t_1 \circ t_2 \\
\Leftrightarrow &\forall t, t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q^m, D', t_1) \cup \mathbb{F}(Q^{i-m}, D', t_2)}) \\
&\quad \wedge t_2 \in Q_2^{i-m}(D'_{\mathbb{F}(Q^{i-m}, D', t_2) \cup \mathbb{F}(Q^m, D', t_1)}) \\
&\quad \wedge t = t_1 \circ t2 \\
\overset{T_{cro}}{\Leftrightarrow} &\forall t, t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \wedge Q_2^{i-m}(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \\
&\quad \wedge t = t_1 \circ t_2 \\
\Leftrightarrow &\forall t: t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)})
\end{aligned}
$$

□

6.2.4 Aggregation Function. For aggregation function, first we use $f(a)$ to denote all aggregate functions such that $Q^{i+1}(D) =_G \gamma_{f(a)}(Q^i(D))$. To prove the rules are correct, we demonstrate that for each group, the rules are correct. Then we show the correctness of two special cases.

PROOF. For a certain group g:

$$
\begin{aligned}
&\forall t: t \in Q^{i+1}(D') \\
\Leftrightarrow &\forall t: t \in_G \gamma_{f(a) \to x}(Q^i(D')) \wedge t.G = g \\
\Leftrightarrow &\forall t, t': t' \in Q^i(D') \wedge t'.G = g \wedge t.x = f(\{\!\{t'.a \mid t' \in Q^i(D') \wedge t'.G = g\}\!\}) \\
\overset{rules}{\Longrightarrow} &\forall t, t': \mathbb{F}(Q^{i+1}, D', t) = \underbrace{\bigcup_{\{\!\{t' \mid t' \in Q^i D' \wedge t'.G = t.G\}\!\}} \mathbb{F}(Q^i, D', t')}_{L} \qquad (T_{agg})
\end{aligned}
$$

$$
\begin{aligned}
&\forall t: t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \wedge t.G = g \\
\overset{T_{agg}}{\Leftrightarrow} &\forall t, t': t \in Q^{i+1}(D'_{\bigcup_L \mathbb{F}(Q^i, D', t')}) \wedge t.G = g \\
\Leftrightarrow &\forall t, t': t \in_G \gamma_{f(a) \to x}(Q^i(D'_{\bigcup_L \mathbb{F}(Q^i, D', t')})) \\
&\quad \wedge t.G = g \wedge t.x = f(\{\!\{t'.a \mid t' \in Q^i(D') \wedge t'.G = g\}\!\}) \\
\overset{HP}{\Leftrightarrow} &\forall t, t': t \in_G \gamma_{f(a)}(Q^i(D')) \wedge \underbrace{t' \in Q^i(D') \wedge t'.G = g}_{A} \\
&\quad \wedge \underbrace{t.G = g \wedge t.x = f(\{\!\{t'.a \mid t' \in Q^i(D') \wedge t'.G = g\}\!\})}_{B} \\
\Leftrightarrow &\forall t: t \in Q^{i+1}(D') \wedge t.G = g
\end{aligned}
$$

For A and B, both of them indicate that the tuples are only computed for group g. It will not affect the results since the it must hold for proof of the certain group g. □

For two special cases:

Case 1 $\mathcal{S}[g] = null \wedge g \notin \mathcal{G} \wedge \Delta = \Delta$. For database D, no tuples will exists for group g of $Q^{i+1}(D)$, and there will be one tuple computed if it runs over updated database D'. Then the following holds:

$$
\begin{cases}
\mathbb{F}(Q^{i+1}, D, t) = \emptyset \wedge t \in g \\
\mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^i, D', t') \wedge t'.G = g
\end{cases}
$$

The rule will generate a tuple for this new group g.

Case 2 $\mathcal{S}[g].d = 1 \wedge \Delta = \Delta$. No tuples will return for $Q^{i+1}(D')$. Then the following holds:

$$
\begin{cases}
\mathbb{F}(Q^{i+1}, D, t) = \mathbb{F}(Q^i, D, t') \wedge t'.G = g \\
\mathbb{F}(Q^{i+1}, D', t) = \emptyset \wedge t \in g
\end{cases}
$$

The rule will propagate nothing since for the updated database, $Q^{i+1}D'$ can compute no tuples.

6.2.5 Union. $Q^{i+1}(D) = Q_1^m(D) \cup Q_2^{i-m}D \wedge m \leq i$

PROOF.

$$
\begin{aligned}
&\forall t: t \in Q_1^m(D') \cup Q_2^{i-m}(D') \\
\Leftrightarrow &\forall t, t_1, t_2: t^{x+y} \in Q^{i+1}(D') \wedge t_1^x \in Q_1^m(D') \wedge t_2^y \in Q_2^{i-m}(D') \\
&\quad \wedge t \in \{\!\{t_1\}\!\} \cup \{\!\{t_2\}\!\} \\
\overset{r_{union}}{\Longrightarrow} &\mathbb{F}(Q^{i+1}, D', t) = \mathbb{F}(Q^m, D', t_1) \cup \mathbb{F}(Q^{i-m}, D', t_2) \qquad (T_{union})
\end{aligned}
$$

$$
\begin{aligned}
&\forall t: t \in Q^{i+1}(D') \\
\Leftrightarrow &\forall t: t \in Q_1^m(D') \cup Q_2^{i-m}(D') \\
\Leftrightarrow &\forall t, t_1, t_2: t_1 \in Q_1^m(D') \wedge t_2 \in Q_2^{i-m}(D') \wedge t \in \{\!\{t_1\}\!\} \cup \{\!\{t_2\}\!\} \\
\overset{HP}{\Leftrightarrow} &\forall t, t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q^m, D', t_1)}) \wedge t_2 \in Q_2^{i-m}(D'_{\mathbb{F}(Q^{i-m}, D', t_2)}) \\
&\quad \wedge t \in \{\!\{t_1\}\!\} \cup \{\!\{t_2\}\!\} \\
\Leftrightarrow &\forall t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q^m, D', t_1) \cup \mathbb{F}(Q^{i-m}, D', t_2)}) \\
&\quad \wedge t_2 \in Q_2^{i-m}(D'_{\mathbb{F}(Q^{i-m}, D', t_2) \cup \mathbb{F}(Q^m, D', t_1)}) \\
&\quad \wedge t \in \{\!\{t_1\}\!\} \cup \{\!\{t_2\}\!\} \\
\overset{T_{union}}{\Leftrightarrow} &\forall t, t_1, t_2: t_1 \in Q_1^m(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \wedge t_2 \in Q_2^{i-m}(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \\
&\quad \wedge t = \{\!\{t_1\}\!\} \cup \{\!\{t_2\}\!\} \\
\Leftrightarrow &\forall t: t \in Q_1^m(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \cup Q_2^{i-m}(D'_{\mathbb{F}(Q^{i+1}, D', t)}) \\
\Leftrightarrow &\forall t: t \in Q^{i+1}(D'_{\mathbb{F}(Q^{i+1}, D', t)})
\end{aligned}
$$

□

7 IMP PROCESSING A WORKLOAD

In this section, we will talk about how the IMP to process a workload.

7.1 Delta Databases

Recall that a workload $\mathcal{W}$ contains a sequence of update operations such that $\mathcal{W} = (U_1, U_2, \ldots, U_n)$. If all the operations are executed, we will have a sequence of database D_S and a sequence of delta database ΔD_S such that:

$$
\begin{cases}
D_S = (D_0, D_1, \ldots, D_n) \\
\Delta D_S = (\Delta D_1, \Delta D_2, \ldots, \Delta D_n)
\end{cases}
$$

To apply incremental procedure to refresh sketches, it requires translate tuples in the delta database to annotated tuples. Thus, for each delta database, a annotated delta database will be created, and a sequence of delta databases $\Delta\mathscr{D}_s$ will be built such that:

$$
\Delta\mathscr{D}_s = (\Delta\mathscr{D}_1, \ldots, \Delta\mathscr{D}_n)
$$

And for each $\Delta\mathscr{D}$, it is created in the same way shown in **??**.

7.2 Queries and Provenance sketches

Since a workload needs to be processed, all queries and provenance sketches of queries need to be considered. Queries will stay the same during processing multiple update operations, which means that the queries keep identical if no queries are added or removed. Thus the provenance sketches of all query in query set needs to be maintained. And the the query set is defined as Q_S which contains all queries that have provenance sketches captured based on current database such that:

$$
Q_S = \{Q_0, Q_1, \ldots, Q_n\}
$$

For all queries that have their sketch created, at certain time point, there exists a set of provenance sketches $\mathcal{PS} = \{\mathcal{P}_0, \ldots, \mathcal{P}_n\}$. If sketches needs to be maintained when an update in the workload, all the sketches created on relations that are changed in the set will be brought up to new versions. Then the sketches set will continuously refresh if all operations are executed sequentially in the workload. Thus, there exists a sequence of provenance sketch sets because of the database changing. We define the provenance sketches sets sequence $\mathcal{PS}_S$ as a sequence of provenance sketches in which each $\mathcal{PS}_i$ contains sketches for queries running over database D_i.

$$
\mathcal{PS}_S = (\mathcal{PS}_0, \mathcal{PS}_1, \ldots, \mathcal{PS}_n)
$$

7.3 State Data

For a sequence of updates to the database, there will be a sequence of state data $\mathcal{S}_s = (\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n)$. $\mathcal{S}_i$ is a set of state data contains all materialized data for queries who need auxiliary information. $\mathcal{S}_0$ is built based on database D_0 and for $i > 0, \mathcal{S}_i$ is obtained through updating $\mathcal{S}_{i-1}$ during incremental processing.

7.4 Incremental Updating Provenance Sketches

For each update operation, the incremental procedure will refresh all the provenance sketches that are created based on relations that are changed in the database. Like the procedure for one sketch of a single query, it still works on the annotated delta database. Other inputs are queries set and state data set. Then it will generate a set of delta provenance sketches that each one is for a query such that:

$$
\mathcal{I}(Q_S, \mathcal{S}_i, \Delta\mathscr{D}_i) = \Delta\mathcal{PS}_i
$$

7.5 Delta Provenance Sketches

The output of incremental procedure is a sequence of delta provenance sketches set if for every event in the workload, the provenance sketches are maintained. Let $\Delta\mathcal{P}_S$ donate the sequence of delta provenance sketch set. The sequence is defined as follows:

$$
\Delta\mathcal{PS}_S = (\Delta\mathcal{PS}_1, \Delta\mathcal{PS}_2, \ldots, \Delta\mathcal{PS}_n) \; where \; \Delta\mathcal{PS}_i = \mathcal{I}(Q_S, \mathcal{S}_{i-1}, \Delta\mathscr{D}_i)
$$

7.6 Constraints

To illustrate the constraint clearly, let us pair $\mathcal{P}_i$ and Q_i together as $\langle Q_i, \mathcal{P}_i \rangle$ so that it is obvious to insight for which query, the sketch is captured. And for the provenance sketch set $\mathcal{PS}_i$, let us use the pair instead of provenance sketch only to better demonstrate the constraints.

The incremental procedure only maintains provenance sketches that already exist and does not create new ones. Then we have the following constrain C_1 as followed:

$$
\forall \langle Q_i, \mathcal{P}_i \rangle \in \mathcal{PS}_{i+1}: \exists (Q_i, \mathcal{P}_i^{'}) \in \mathcal{PS}_i \qquad (C_1)
$$

$$
\forall \langle Q_i, \mathcal{P}_i \rangle, \langle Q_j, \mathcal{P}_j \rangle \in \mathcal{PS}_l: Q_i = Q_j \rightarrow \mathcal{P}_i = \mathcal{P}_j
$$

8 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.

9 IN-MEMORY IMPLEMENTATION

We have implemented the incremental approach(IMP) to maintain provenance sketches for queries in a middleware on top of DBMS. Fig. 2 demonstrate an overview of our middleware. Our IMP engine is the pipeline of color red. To obtain the latest provenance sketch for a query, the user will provide the query and delta database information (update statements or delta relations). The IMP engine will execute the procedure $\mathcal{I}(Q, \mathcal{S}, \Delta D)$ to generate the delta sketches. And during maintaining the sketches, stored state data will be fetched from the database. At last, new provenance sketches will be computed by calculating the original and the delta sketches.

9.1 Data Chunk and Column Chunk

All input and output data of an operator is stored in a columnar representation, which means that data in a column of same datatype is kept in a vector. Then all these vectors are store in a data chunk with metadata. There are two types of data chunks: inserted data chunk and deleted data chunk, which stores annotated tuples of insertion or deletion respectively. The data model for our incremental processing is to associate each tuple with its provenance sketch. When accessing the delta relations, the IMP engine processes each tuple and attaches the fragment information to it and stores fragments in its own vector. For each operator in a query, the engine uses the data chunks propagated up by its child(ren), and generates data chunks with only designated columns. Column chunk stores data with meta information in columnar fashion as well which is used for expressions evaluation. The advantage of using columnar fashion is mainly because of efficiency to handle data. For expressions evaluation, the engine can process data of columnar caching in memory continuously without redirecting to elements if stored in a row-oriented style.

9.2 State Data

State data of each operator is to best exploit specialized data structures that are suited best for operations for data annotated with sketches. When maintaining sketches for a query, the engine will first check if this query has met before to know weather to build state data or fetch the materialized state data. The state data will be updated during maintaining the sketch and be retained in the database (the pipeline of bottom in Fig. 2) when the engine does not maintain its query anymore.

9.3 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. At the beginning of each batch we initialize an empty copy of the state that is used to track which groups have been updated as part of the current batch. Whenever a group is updated for the first time in a batch (there is no entry for this group in the copy yet), then we copy the current state for the group to the per-batch data structure. The purpose of this data structure is to track which groups were affected by a batch and cache their previous state. We do not output any delta results while processing a batch. Instead after processing the batch, we use the per batch data structure to determine which groups have been updated and output deltas for the group as described in Sec. 5.6 and 5.7. Note that we do not copy $\mathcal{F}_g$ to the per batch data structure as we do not need the previous state for this data structure. We follow the same approach for the top-k operator.

9.4 Optimizations

Data transfer between IMP client and DBMS can reduce the efficiency of the performance of incremental procedure. To overcome this issue, we have optimized operators that need to access database *i.e.*, table accessing and join operators. Both selection-push-down and bloom filter techniques target to reduce the amount of data in transferring. Less data in transferring, less overhead increasing for incremental procedure.

9.4.1 Bloom Filters For Join. For join operators, it needs the database engine to compute the joinabl partners for deltas of join. For a table joined another one with low selectivity, most tuples do not have join partners, and if those tuples can be removed up-front and not transfer to the database, then it will reduce the overhead of join performance. To pre-decide which tuples can potentially pair others, bloom filter can select them out. We build for each attribute in equal join conditions a bloom filter to tell the existence of values of in the other side of join. Before passing delta tuples into database, first check each tuple if there exists tuples can matching it. Then send the tuples passing bloom filter validation to the database and get back the result. Bloom filters is materialized locally and used for subsequently updates.

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

9.4.3 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 aciveve 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 alway the current minimum/maximum and the new minimum/maximum will always be in the state data if it comes from inputs.

10 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 were run on 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.

Pengyuan says: Variance: tpch: max:0.016908437526766667, Crime: 3.7284266666666774e-08, Sync_small: 0.052577969784711104, sync_large: 3.459550946554321

10.1 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 10M rows tables where each one contains at least 12 attributes with different datatypes. There is a key attribute (unique) for each table. For other attributes, we control the size of the domain of one attribute(a) and sample uniformly from the domain. Then, we generate the rest of the attributes that each attribute has a correlation with the attribute a. We vary the following characteristics of the generated data: 1. the number of distinct values per attribute which is for: a. number of groups for group-by queries, b. queries of M-N joins. 2. for join selectivity experiments, we also vary the joinable attribute value and control the range of values that are joinable.

Baseline. For the maintenance of provenance sketches, we evaluate the performance of our IMP framework called incremental maintenance against a baseline where the provenance sketches are captured from scratch called full maintenance.

Parameter. We vary the delta size (the amount of delta tuples that are inserted or deleted) in all our experiments in two categories: 1. realistic delta size where the delta size is 10, 50, 100, 500 and 1000; 2. large delta size where the delta size gradually increase from 0.1% of the updated table to 10% with 0.1% of each incremental.

10.2 TPC-H

We use TPC-H[4] benchmark at SF1 ($\sim$ 1 GB) and SF10 ($\sim$ 10GB) to evaluate the incremental against full maintenance using the delta with realistic size. The queries we evaluated have the following properties: 1. Provenance sketches are beneficial to use to answer queries [27]; 2. Queries have complex structures: multi-way joins, large number of aggregation functions and top-k operators.

In these experiments, we utilized the selection push down and join bloom filter optimizations (see Sec. 9.4). Sec. 10.2 and Sec. 10.2 show the runtime of incremental vs. full maintenance for SF1 and SF10, respectively, using the realistic delta size. Incremental maintenance outperforms full maintenance by at least a factor of 3.9 and up to a factor of 2497.

In general, the joins in our IMP engine are more expensive. As our engine uses the database to execute $\Delta R \bowtie S$, incrementally maintaining joins requires data transfer for all delta. The more join operators a query contains, the higher cost for IMP engine to complete the incremental procedure. The TPC-H $Q5$, $Q7$ and $Q9$ have the least benefit because all these three queries contain 6-way join. While $Q4$, $Q12$, $Q14$, $Q19$ have a relative better performance than $Q5$, $Q7$ and $Q9$ since these queries only contains 2-way joins. $Q1$ and $Q6$ have the best performance because these two queries only work on a single table without join.

Sec. 10.2 shows the incremental maintenance runtime for both insertion and deletion for certain amount of delta for 10 GB database size.

10.3 Crime

The Crime[5] dataset records incidents in Chicago and consists of a single 1.87GB table with $7.3M$ rows. We use two queries(SQL code for the queries are shown in Appendix A): `CQ1`: The numbers crime of each year in each beat (geographical location). `CQ2`: Areas with more than 1000 crimes. Sec. 10.3 shows the runtime of full vs. incremental approach to update provenance sketches under the realistic delta size. Sec. 10.3 show the incremental maintenance runtime for both insertion and deletion under given delta size.

10.4 End-to-end Experiments

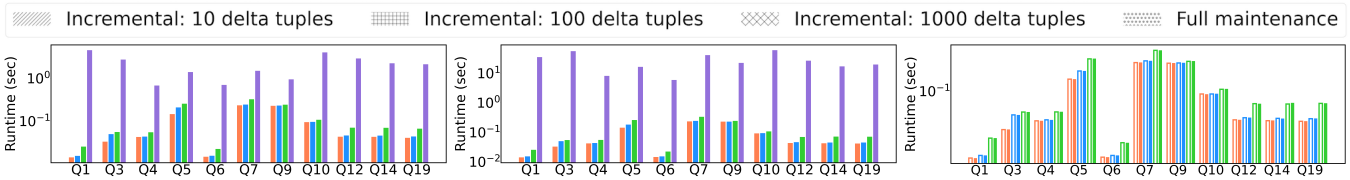
In this experiment, we evaluate our approach on a mixed workload consisting of queries and updates and measure the end-to-end runtime. For the workload, there are total 1000 operations. We control the ratio between query and update called query-update ratio.

There are two baselines in this experiment: queries execution without using provenance sketches called non-sketch approach and queries using sketches but capturing provenance sketches from scratch whenever they are needed called full-use approach. And our approach called incremental-use approach where the provenance sketches are maintained incrementally using IMP engine.

[27] presents the technique of reusing sketches for similar queries with the same structure at different parameter settings, which allow us to use provenance sketches of Q_1 to answer Q_2. In this experiment, we apply this technique: using existing sketches to answer queries whenever it is applicable, while when there is no existing sketches

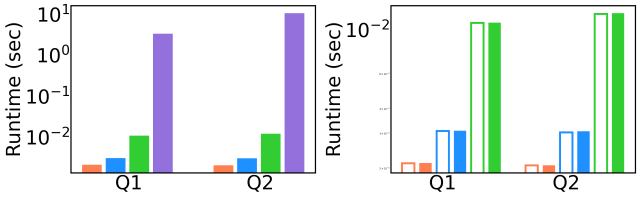
[4]TPC-H is a Decision Support Benchmark: www.https://www.tpc.org/tpch/

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



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

Figure 7: TPC-H Benchmark



(a) Crimes (b) Insertion and deletion

Figure 8: Crimes

that we can use to answer a query, the provenance sketch of this query will be captured.

For this experiment, we utilize the realistic delta size: 1, 20, 200 and 2000. We vary the query-update ratio as: 1U1Q (one update per one query), 1U5Q (one update per five queries) and 5U1Q (five updates per one query). For non-sketch approach, we directly run the queries or updates. For full-use approach, we always capture sketches to answer the queries when encountering queries. For incremental-use approach, we capture the provenance sketches the first time queries are encountered. 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 are to cache the delta and they are associated with an update version that allows us to fetch delta between the last maintained version of database and the current version of database.

Fig. 9 shows the runtime under different combinations of query-update ratios and delta sizes. All the experiments show that the full-use approach leads to the highest cost because each time we need to use sketches to answer a query, we have to generate the sketches from scratch after update operations(there is no need generating the provenance sketches for queries that can reuse sketches already existing between two updates). In general, non-sketch approach can outperform the full-use approach while it is worse then incremental-use approach. The incremental-use approach can have the best performance among the three only except for 5U1Q with each update containing 2000 tuples, because most of the operations are updates and each incremental maintenance will process more delta accumulating by update operations between two maintenances.

10.5 Microbenchmarks

To evaluate different aspects of our approach response to the changing of the parameter (delta size), we use synthetically generated data to control the properties of the queries. The experiments over synthetic data are in the following categories: 1. different numbers of aggregation functions, 2. fixed number of aggregation functions with different groups, 3. $M - N$ joins, 4. joins with different selectivity, 5. different provenance sketch sizes. We compare the performance of incremental maintenance against full maintenance as a baseline

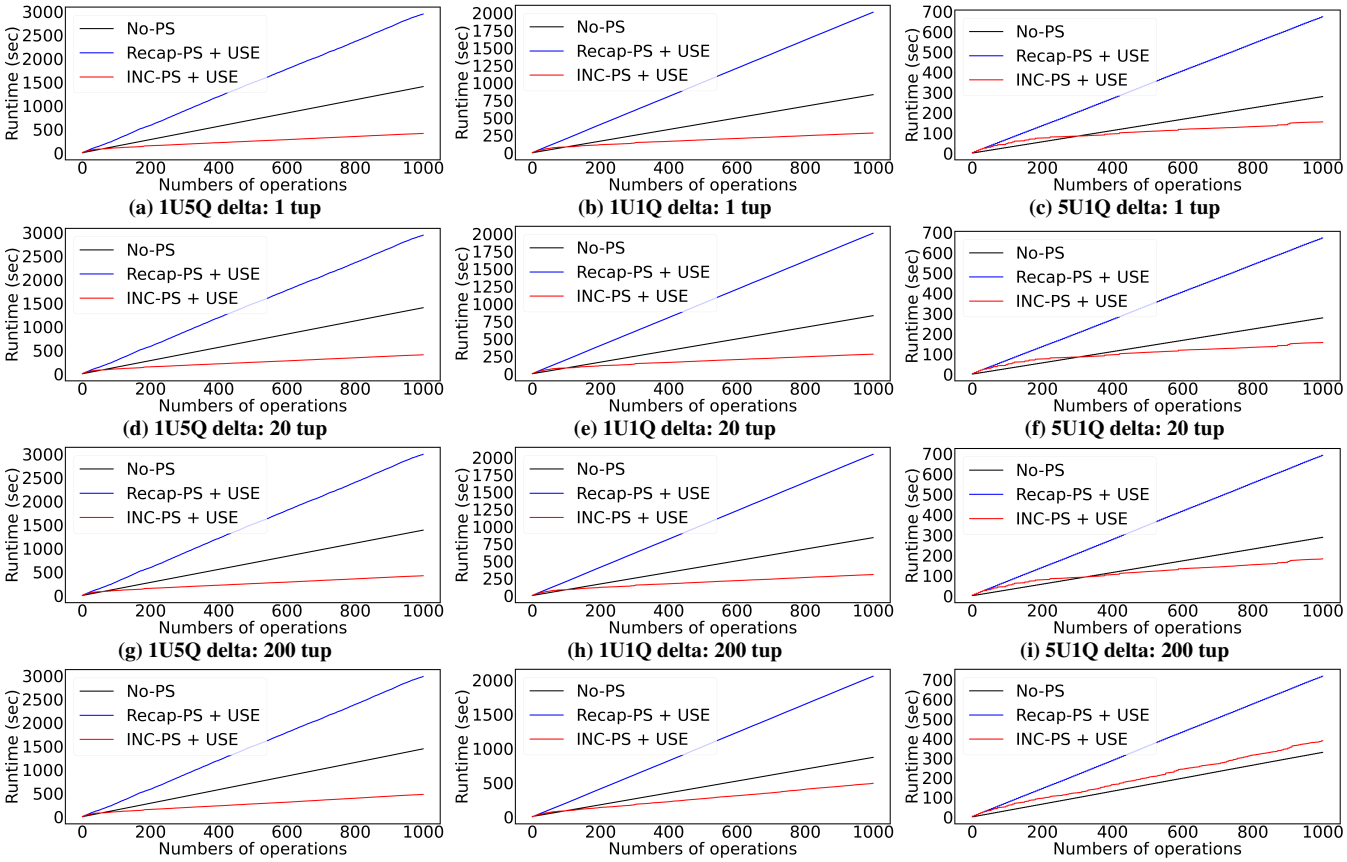
For incremental maintenance, we examine each category using both realistic delta size and large delta size: 0.1% of original table size to 10% (categories 1 and 2) or 1.5% (categories 3, 4, 5, and 6) of original table size with 0.1% of each incremental.

Number of groups. We use a query template group-by-aggregation without join Q_{groups} (SQL code for the query is shown in Appendix A) which has a fixed number of aggregation functions to evaluate the performance of IMP varying the number of groups in : 50, $1K$, $5K$ and $500K$.

As the data structures maintained by our approach for aggregation store an entry for each group and the number of groups affected by delta will also depend on the number of groups, we expect that runtime will increase when increasing the number of groups. As shown in Fig. 10b, for delta sizes up to 1000 tuples, incremental maintenance outperforms by 2 (500k groups) to 3 (50 groups) orders of magnitude. Fig. 11b shows that the break even point (where full maintenance starts to outperform incremental maintenance) lies at delta sizes between ~2.5% (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 template Q_{having} (SQL code for the query is shown in Appendix A) 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 for aggregation functions are 5000. We vary the number of aggregation functions used in the `HAVING` condition as our approach to maintain the results for these aggregation functions and compare the performance of our incremental maintenance against full approach. Fig. 10a and Fig. 11a show the runtime of incremental vs. full maintenance using both realistic delta size and large delta size. The runtime of incremental maintainer is linear in the size of the delta for this query template with a coefficient that is dependent on the number of aggregation function. For realistic delta size, incremental maintenance outperforms full maintenance by ~ 2 orders of magnitude. As shown in Fig. 11a, incremental maintenance is faster than full maintenance for deltas of up to ~ 5% of the database.

Join. We evaluate the performance of aggregation-group-by-having queries over the result of a an equi-join using query template Q_{join} (SQL code for the query is shown in Appendix A). Both joined table have $10M$ rows. The synthetic tables are designed as the follows: for a $m - n$ join of $R \bowtie S$, the selectivity is 100% for table S, and there are $10^8/n$ distinct join attribute values with a multiplicity of n; for



(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 9: End to end experiments

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.

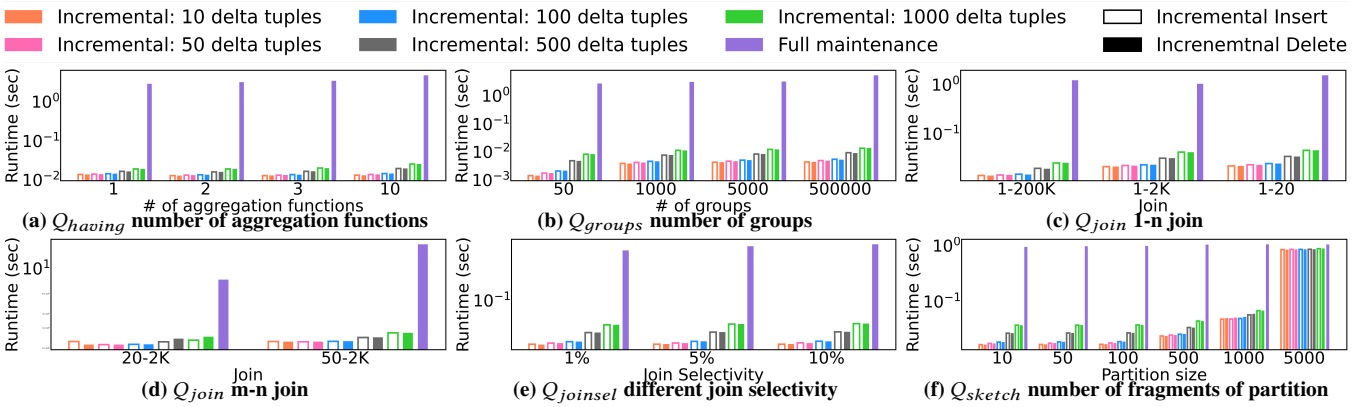
Fig. 10c and Fig. 11c show the runtime of incremental vs. full maintenance for $1-n$ joins, and Fig. 10d and Fig. 11d show the performance of both full and incremental approaches for $m-2K$ joins.

For the 1-n join, the 1-20 join is more expensive than 1-20k and 1-200k joins because even 1-20 join has less join result tuples, but since there are more groups for the aggregation functions above join operator (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 have the same number of groups for aggregation operation, and the 50-2k will have more join results to process which leads to more expensive than a 20-2k join.

Recall in Sec. 10.2, our IMP engine depends on the database to compute $\Delta R \bowtie S$, incrementally maintaining joins requires data transfer for all delta tuples. Thus, 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 evaluate this optimization in Sec. 10.6).

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) (SQL code for the query is shown in Appendix A) to evaluate different join selectivity: 1%, 5%, and 10%. We control the join attribute's value in the tables S (this attribute joins with another attribute in table R) such that a certain percentage of value can be joined. Fig. 10e and Fig. 11e show the runtime of incremental vs. full maintenance using both realistic delta size and large delta size. Higher selectivity means that it is more expensive for IMP engine to complete the maintenance procedure. Because for a fixed delta size, the higher the selectivity is, the more the tuple in join result are, which increases time for the database engine to compute and for data to transfer.

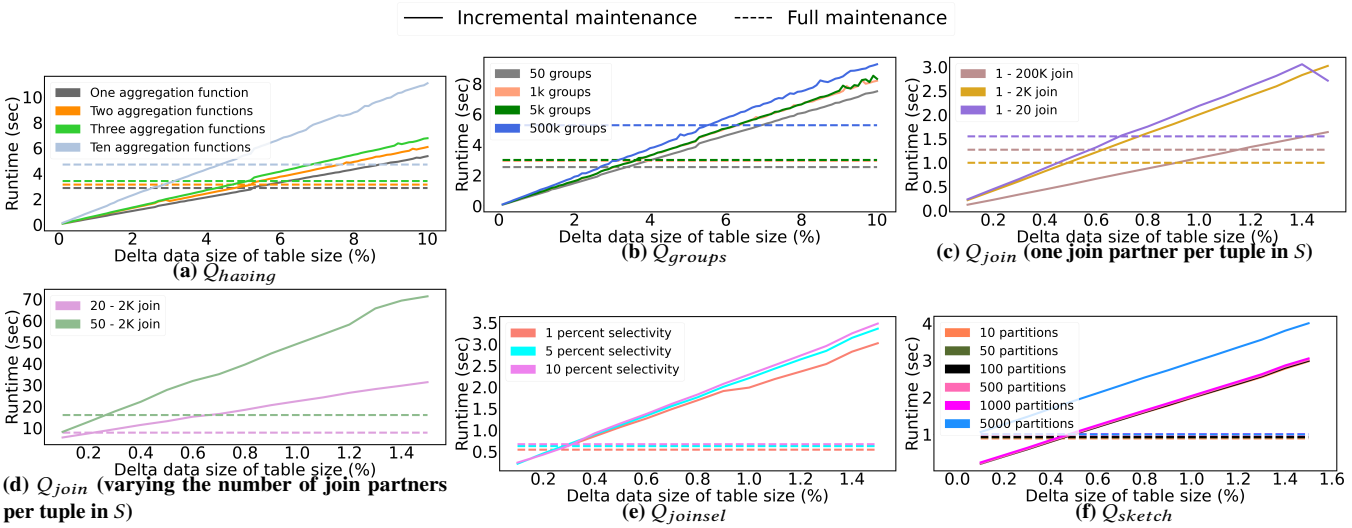
Varying Partition Granularity. In this experiment, we vary #frag, the number of fragments of the partition on which provenance sketches are based on, to examine the performance of our incremental approach. We use query template Q_{sketch} (SQL code for the query is shown in Appendix A) which is aggregation-group-by over join query to evaluate the performance for both incremental and full maintenance by varying the number of fragments of partition for attributes used to build the provenance sketches from 10 to 5000. Fig. 10f and Fig. 11f show the runtime for incremental and full maintenance for both realistic and large delta size. While the cost of full maintenance is also impacted by #frag, the dominating cost is



(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 10: 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 the number of join partners per tuple in S) (e) $Q_{joinsel}$ (f) Q_{sketch}

Figure 11: Varying delta size to determine the "break even point", i.e., up to which delta size does incremental maintenance outperform full maintenance.

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

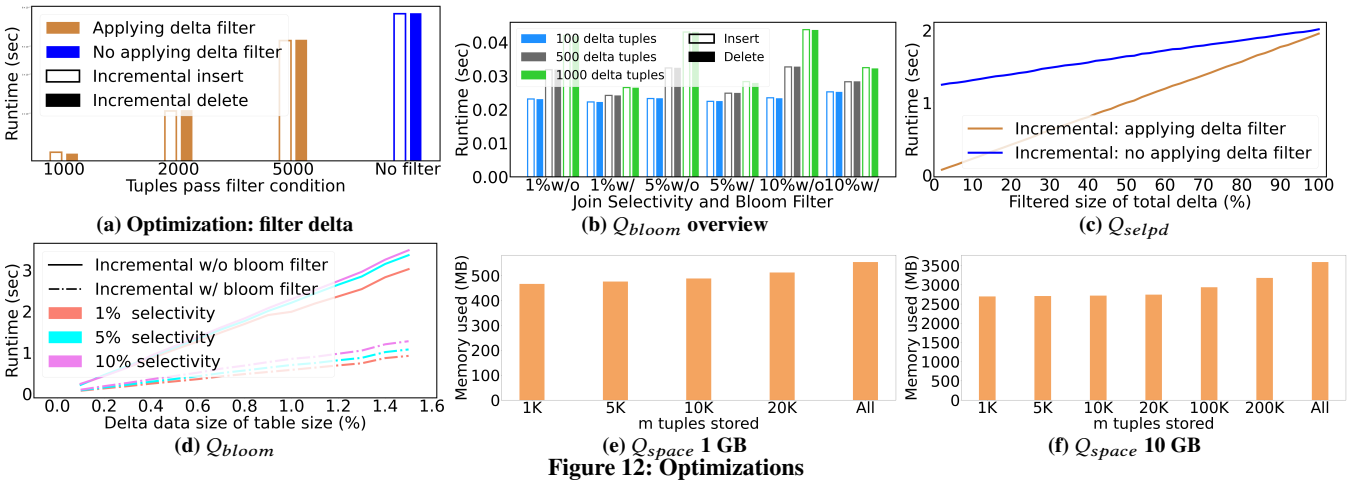
In this section, we evaluate the performance of our IMP engine to maintain sketches for queries having different properties response to the varying of delta size. The results present that the incremental maintenance outperform full maintenance for delta of realistic size.

10.6 Optimizations

To understand the behavior of our optimizations, we use the synthetic dataset to evaluate the performance of IMP engine with both realistic and large delta size. We only examine the performance of incremental approach using our IMP engine.

Filtering deltas through selection push down. In this experiment, we evaluate the effectiveness of our selection-push-down optimization that filters delta based on selection conditions of the query for which we are maintaining a sketch. We use the query template Q_{selpd} which is a group-by-aggregation without join query(SQL code for the queries are shown in Appendix A). We evaluate the performance of incremental maintenance with and without selection-push-down, varying the selectivity of the selection condition (`WHERE` clause) of the query. We fix the delta size to 2.5% of the table. Then we gradually increase the fraction of the data that fulfills the filter condition from 2% to 100%. Fig. 12c shows runtime of incremental maintenance with and without the selection-push-down optimization. Note that the x-axis shows the percentage of data of total size that can pass the filter conditions. The results show that the runtime of filtering deltas increases linearly in the selectivity of the query's selection condition. Even for large selectivity, the cost of filtering delta tuples is amortized by reducing the cost of incremental maintenance. Thus, this optimization should be applied whenever possible.

Join optimization using bloom filter. Another optimization we applied in IMP engine is to use bloom filter for each join to track which tuples potentially have join partners. This enables us to avoid evaluating a join $\Delta R \bowtie S$ ($R \bowtie \Delta S$) using the database when we can determine based on the bloom filter that no tuples in the delta have



(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 12: Optimizations

any join partners or at least reduce the delta size. To evaluate this optimization, we use the query template $Q_{joinsel}$ (SQL code for the queries are shown in Appendix A) which is a group-by-aggregation over join query. Fig. 12b shows the runtime of incremental approach applying bloom filter for joins with different selectivity given realistic delta size. Fig. 12d shows runtime of incremental maintenance with and without bloom filter for large delta size increasing from 0.1% to 1.5% maintenance. The result shows that pre-deciding the joinable part in delta can reduce the overhead of incremental processing due to 1) small amount data transferring between IMP and database, 2) less time cost to compute the join result inside database engine.

In general, these two optimizations are effective by reducing the unnecessary delta for the IMP engine to maintain sketches. Both optimizations can improve the performance of IMP engine when incrementally maintaining the provenance sketches. **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. 12e and Fig. 12f 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.

11 TRANSACTION AND DELTA DATA

11.1 Computing Delta Data

The IMP will use delta data to compute the delta provenance sketch for queries. Everytime 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`.

11.2 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 vertion v2 using delta data from the difference between v1 and v2 of the database, and then apply sketches and database of v2.

12 COST ESTIMATION

Pengyuan says: Since this part is not much related to this paper, we can delete most of the content here, just keeping the result and give a brief descriptions. And move the experiment result to EXP section.

[27] has studied the cost of capture provenance sketches and benefits from running queries with sketches. In this work, we learn how incremental maintenance performs under database update. Next it is import to get a knowledge of the performance for queries with and without provenance sketches under database updates. In this section, we will have a brief discuss about a linear model for cost estimation.

12.1 Cost-based Computation

Let us start with the databases in time-series fashion: there are $n+1$ versions of databases with each one D_i corresponding to a time point T_i, with delta between D_i and D_{i-1} : ΔD_i, the provenance sketch $\mathcal{P}_i$ and total ${R^{Q}}^{\#}{}_i$ times running a query Q, as shown below.

$TimePoint:$	T_0	$\cdots$	T_i	$\cdots$	T_n
$Database:$	D_0	$\cdots$	D_i	$\cdots$	D_n
$Delta:$	ΔD_0	$\cdots$	ΔD_i	$\cdots$	ΔD_n
$ProvenanceSketch:$	$\mathcal{P}_0$	$\cdots$	$\mathcal{P}_i$	$\cdots$	$\mathcal{P}_n$
$QueryExecutionTimes:$	${R^{Q}}^{\#}{}_0$	$\cdots$	${R^{Q}}^{\#}{}_i$	$\cdots$	${R^{Q}}^{\#}{}_n$

Let us focus on one time points T_i. At this point, if the query Q needs to be executed, there are mainly two different ways: running query with and without provenance sketches. 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 at time point T_i. For queries running with sketches, we generally 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.

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

12.2 Cost Estimation

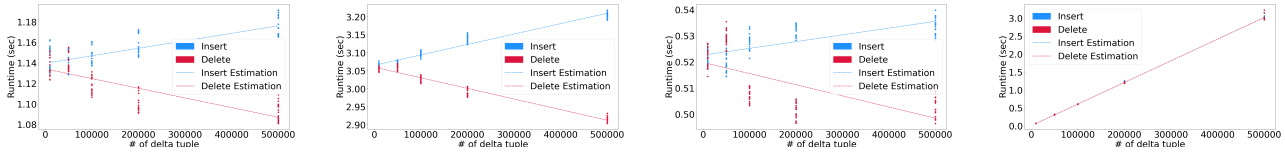
In this part, we will discuss our linear model for running queries without provenance sketch, capturing sketches, incrementally updating sketchs and using sketches for queries based on delta size. For this linear model, if we can decide the the cost for each one based on delta size, then, we can decide the way to run the queries: using sketch or not, and if using sketch, how to update current sketches based on size of delta database.

In our experiment, we evaluate one query Q_{having}. The delta size (delta tuple number) are: 10000, 50000, 100000, 200000 and 500000 out of 10M rows of original table. Fig. 13 shows the performance of the query use or not use sketch and how the sketches are refreshed according to the delta size. And in each figure, we plotted the cost for insertion and deletion under the same delta size. The dots in each figure show the actual running cost under certain delta size, and the dash line is a prediction based on these actual running. Note that for incremental figure, the insert and delete estimations are overlap.

We also did end-to-end experiments to evaluate queries running with and without sketches, as well as how to update the sketches when database changes if queries running with sketches. In this experiments, we also test queries that can reuse sketch from other queries. For each queries, we evaluated small delta size (10 rows and 1K rows), medium size(100K rows, 1% of table size) and large size (1 M rows, 10% of table size). Under each delta size, we test different queries running times, and how frequent the database update.

To define the update frequency of the database, we assume that each query will be executed in a same short period, which means that the time between two executions of the same queries will be the same. And we assume that during two state of a database, it is frequent update if queries can only execute at most 10 times, it is a medium frequent update if the execution times are between 10 and 50 times, and otherwise it is less frequent updated database when execution times are more that 50. Thus, for a database never updated, we will have no delta database to deal with.

?? shows the performance of a QHaving query executed under different update frequencies. **??** and **??** shows performance for queries the same setting as **??** and using the same sketches as QHaving query. All the results show that if the database is not frequently updated, running queries with sketches can be the best choice. And if the database is refreshed very frequently, the choice to run queries with



(a) Q_{having} **no-ps** **(b)** Q_{having} **capture** **(c)** Q_{having} **use ps** **(d)** Q_{having} **incremental update**

Figure 13: Cost estimation

and without sketch depends. And for queries execution using sketch, if the delta size if small, an incremental maintenance of sketches is better than capture sketch again.

?? and **??** show the results of join queries with 5% and 1% respectively. **??** and **??** show the queries running using the same sketch as that of QJoin_5%. **??** and **??** show performance of queries using the same sketch as QJoin_1% does. In these experiments, since the join selectivities are relative small. Queries running without sketch have good performance.

Pengyuan says: end to end plots should be here or somewhere earlier

13 CONCLUSIONS AND FUTURE WORK

We present in-memory incremental maintenance of provenance sketches (IMP), an approach to maintain provenance sketches incrementally when the database changes. Our in-memory engine can efficiently produce the delta sketches and compute the new provenance sketches to correctly reflect which parts of the updated database contain provenance information to answer the queries. With optimization using bloom filter and selection-push-down, it can further improve the performance of the incremental process. The experimental evaluation demonstrates the effectiveness of our approach and optimizations.

In the future, we will investigate how

- Improve IMP engine to support wider range of queries.
- Investigate approaches to further optimize operators to achieve more efficient maintenance.
- Study different cost estimation models. Get concrete strategies for online fashion queries running with database update between using and not using provenance sketches and how to maintain sketches.
- Zhen's comments: Since for relational database, the relational algebra is not expressive enough and it lacks flexibility to use specialized data structures to store operator state. In the future, it is necessary to exploit and maintain the sketches in object database as JSON datatype.
- Zhen's comments: Consider to exploit the provenance related work over vector data and develop maintenance for vector data.

REFERENCES

[1] Martín Abadi, Frank McSherry, and Gordon D. Plotkin. Foundations of differential dataflow. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034, pages 71–83, 2015.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505. Morgan Kaufmann, 2000.

[3] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. Gprom - A swiss army knife for your provenance needs. *IEEE Data Eng. Bull.*, 41(1):51–62, 2018.

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

[5] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 61–71. ACM Press, 1986.

[6] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. Dbsp: Automatic incremental view maintenance for rich query languages. *PVLDB*, 16(7):1601–1614, 2023.

[7] Peter Buneman and Eric K. Clemons. Efficiently monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, 1979.

[8] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 577–589. Morgan Kaufmann, 1991.

[9] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200. IEEE Computer Society, 1995.

[10] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 469–480. ACM Press, 1996.

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

[12] Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. Implementation of delayed updates in heraclitus. In Alain Pirotte, Claude Delobel, and Georg Gottlob, editors, *Advances in Database Technology - EDBT'92, 3rd International Conference on Extending Database Technology, Vienna, Austria, March 23-27, 1992, Proceedings*, volume 580 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 1992.

[13] Boris Glavic, Renée J. Miller, and Gustavo Alonso. Using SQL for efficient generation and querying of provenance information. In Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael P. Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, volume 8000 of *Lecture Notes in Computer Science*, pages 291–320. Springer, 2013.

[14] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 328–339. ACM Press, 1995.

[15] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In Kotagiri Ramamohanarao, James Harland, and Guozhu Dong, editors, *Proceedings of the Workshop on Deductive Databases held in conjunction with the Joint International Conference and Symposium on Logic Programming, Washington, D.C., USA, Saturday, November 14, 1992*, volume CITRI/TR-92-65 of *Technical Report*, pages 185–194. Department of Computer Science, University of Melbourne, 1992.

[16] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[17] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 157–166. ACM Press, 1993.

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

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

[20] Volker Küchenhoff. On the efficient computation of the difference between concecutive database states. In Claude Delobel, Michael Kifer, and Yoshifumi Masunaga, editors, *Deductive and Object-Oriented Databases, Second International Conference, DOOD'91, Munich, Germany, December 16-18, 1991, Proceedings*, volume 566 of *Lecture Notes in Computer Science*, pages 478–502. Springer, 1991.

[21] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In Sharad Mehrotra and Timos K. Sellis, editors, *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 307–318. ACM, 2001.

[22] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 476–487. Morgan Kaufmann, 1998.

[23] Haneen Mohammed, Charlie Summers, Sughosh Kaushik, and Eugene Wu. Smokedduck demonstration: Sqlstepper. In Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia, editors, *SIGMOD*, pages 183–186, 2023.

[24] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 1560–1568. AAAI Press, 2015.

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

[26] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, 2016.

[27] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. Provenance-based data skipping. *Proc. VLDB Endow.*, 15(3):451–464, 2021.

[28] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. Heuristic and cost-based optimization for diverse provenance tasks. *IEEE Trans. Knowl. Data Eng.*, 31(7):1267–1280, 2019.

[29] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, and Venkatesh Radhakrishnan. Provenance-aware query optimization. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 473–484. IEEE Computer Society, 2017.

[30] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 802–813. Morgan Kaufmann, 2002.

[31] Fotis Psallidas and Eugene Wu. Smoke: Fine-grained lineage at interactive speed. *CoRR*, abs/1801.07237, 2018.

[32] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql: Provenance and probability management in postgresql. *Proc. VLDB Endow.*, 11(12):2034–2037, 2018.

[33] Oded Shmueli and Alon Itai. Maintenance of views. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 240–255. ACM Press, 1984.

[34] Dimitra Vista. View maintenance in relational and deductive databases by incremental query evaluation. In John E. Botsford, Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Kelly A. Lyons, Jacob Slonim, and J. Howard Johnson, editors, *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 3, 1994, Toronto, Ontario, Canada*, page 70. IBM, 1994.

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

[36] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 615–626. ACM, 2010.

[37] 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. Recommending materialized views and indexes with IBM DB2 design advisor. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 180–188. IEEE Computer Society, 2004.

A APPENDIX I: QUERY LIST

A.1 Synthetic dataset query list

A.1.1 Different number of aggregation functions.

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
```

A.1.2 Aggregation functions and group numbers.

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
```

A.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
```

A.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
```

A.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
```

A.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
```

A.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
```