Provenance-based Data Skipping (TechReport)

Xing Niu*, Ziyu Liu*, Pengyuan Li*, Boris Glavic*, Dieter Gawlick$^{\alpha}$, Vasudha Krishnaswamy$^{\alpha}$,
Zhen Hua Liu$^{\alpha}$, Danica Porobic$^{\alpha}$
Illinois Institute of Technology*, Oracle$^{\alpha}$
{xniu7,zliu102,pli26}@hawk.iit.edu,bglavic@iit.edu
{dieter.gawlick,vasudha.krishnaswamy,zhen.liu,danica.porobic}@oracle.com

ABSTRACT

Database systems analyze queries to determine upfront which data is needed for answering them and use indexes and other physical design techniques to speed-up access to that data. However, for important classes of queries, e.g., HAVING and top-k queries, it is impossible to determine up-front what data is *relevant*. To overcome this limitation, we develop provenance-based data skipping (PBDS), a novel approach that generates provenance sketches to concisely encode what data is relevant for a query. Once a provenance sketch has been captured it is used to speed up subsequent queries. PBDS can exploit physical design artifacts such as indexes and zone maps.

PVLDB Reference Format:
Xing Niu*, Ziyu Liu*, Pengyuan Li*, Boris Glavic*, Dieter Gawlick$^{\alpha}$, Vasudha Krishnaswamy$^{\alpha}$, Zhen Hua Liu$^{\alpha}$, Danica Porobic$^{\alpha}$. Provenance-based Data Skipping (TechReport). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Physical design techniques such as index structures, zone maps, and horizontal partitioning have been used to provide fast access to data based on its characteristics. To use any such data structure to answer a query, database systems statically analyze the query to determine what data is *relevant* for answering it. Based on this information the database optimizer (i) optimizes the query to filter out irrelevant data as early as possible (e.g., using techniques like selection-pushdown) and (ii) determines how to execute this filtering step efficiently.

EXAMPLE 1. *Query Q_1 shown in Fig. 1a returns the population density (*`popden`*) of cities in California. The database system may use an index on column* `state`*, if it exists, to identify cities in California, reducing the I/O cost of the query.*

While this approach of statically analyzing a query to determine a declarative description of what data is relevant is effective for some queries, it is often not possible to determine relevance statically.

EXAMPLE 2. *Query Q_2 shown in Fig. 1a returns the state with the highest average city population density. The result of this query over an example database (Fig. 1b) is shown in Fig. 1d. CA has the highest average population density. Thus, only the 2^{nd} and 3^{rd} tuple are needed to produce the result. One possible declarative description of the relevant inputs is* `state = 'CA'`*. However, unlike the previous example, this description is* data-dependent*. For instance, if we delete the 5^{th} row, then NY has the highest average density and* `state = 'CA'` *no longer correctly describes the relevant inputs.*

Q_1

```sql
SELECT city, popden
FROM cities
WHERE state = 'CA';
```

Q_2

```sql
SELECT state, avg(popden) AS avgden
FROM cities
GROUP BY state
ORDER BY avgden DESC
LIMIT 1;
```

$Q_2[\mathcal{P}_{state}]$

```sql
SELECT state, avg(popden) AS avgden
FROM cities
WHERE state BETWEEN 'AL' AND 'DE'
GROUP BY state
ORDER BY avgden DESC
LIMIT 1;
```

(a) Queries

	popden	city	state	
t_1	4200	Anchorage	AK	f_1
t_2	6000	San Diego	CA	
t_3	5000	Sacramento	CA	
t_4	7000	New York	NY	f_3
t_5	2000	Buffalo	NY	
t_6	3700	Austin	TX	f_4
t_7	2500	Houston	TX	

(b) cities relation

city	popden
San Diego	6000
Sacramento	5000

(c) Result of Q_1

state	avgden
CA	5500

(d) Result of Q_2 (and $Q_2[\mathcal{P}_{state}]$)

f_1 = [AL,DE], f_2 = [FL,MI], f_3 = [MN,OK], f_4 = [OR,WY]

g_1 = [1000,4000], g_2 = [4001,9000]

(e) A range partition of relation cities on attribute `state` (F_{state}, top) and `popden` (F_{popden}, bottom)

Figure 1: Running Example

Even though the query in the example above is selective, state-of-the-art systems are incapable of exploiting this selectivity since it is impossible to determine a declarative condition capturing what is relevant by static analysis. In fact, there are many important classes of queries including top-k queries and aggregation queries with **`HAVING`**, for which static analysis is insufficient to determine relevance. Thus, while these queries may be quite selective, databases fail to exploit physical design artifacts to speed up their execution.

To overcome this shortcoming of current systems and better utilize existing physical design artifacts, we propose to analyze queries at runtime to determine concise and declarative descriptions of what

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

data is relevant (sufficient) for answering a query. We use the provenance of a query to determine such descriptions since for most provenance models, the provenance of a query is *sufficient* for answering the query [40]. That is, if we evaluate a query over its provenance this yields the same result as evaluating the query over the full database. We introduce *provenance sketches* which are concise descriptions of supersets of the provenance of a query. Given a horizontal partition of an input table, a provenance sketch records which fragments of the partition contain provenance. Since only some tuples of a fragment may be provenance, any provenance sketch encodes a superset of the provenance of a query. Similar to query answering with views, a sketch captured for one query is used to speed-up the subsequent evaluation of the same or other queries.

EXAMPLE 3. *Using Lineage [25, 40], the provenance of query Q_2 from Ex. 2 is the set of the tuples highlighted in yellow in Fig. 1b. Consider a provenance sketch based on a range-partition of the input on attribute* `state` *which is shown in Fig. 1e. We assume that states are ordered lexicographically, e.g.,* `CA` *belongs to the interval* [AL,DE]. *In fig. 1b we show the fragment that each tuple belongs to on the right. The sketch $\mathcal{P}_{state}$ of Q_2 according to this partition is* $\{f_1\}$, *i.e., f_1 is the only fragment that contains provenance.*

Creating Provenance Sketches. We present techniques for instrumenting a query to compute a provenance sketch that are based on annotation propagation techniques developed for provenance capture (e.g., [14, 16, 42]). Our approach has significantly lower runtime and storage overhead than such techniques. Given range-partition(s) for one or more of the relations accessed by a query, we annotate each input row with the fragment it belongs to. These annotations are then propagated to ensure that each intermediate result is annotated with a set of fragments that is a superset of its provenance. We use bitvectors to compactly encode sets of fragments and develop fast methods for creating and merging these bitvectors. These optimizations can be implemented using common database extensibility mechanisms.

Using Provenance Sketches. Once a provenance sketch for a query Q has been created, we would like to use it to speed-up the subsequent execution of Q or queries similar to Q. For that we need to be able to instrument a query to restrict its execution to data described by the provenance sketch. This can be achieved by filtering out data not belonging to the sketch using a disjunction of range restrictions.

EXAMPLE 4. *Consider the sketch $\mathcal{P}_{state} = \{f_1\}$ from Ex. 3. It describes a superset of what data is relevant for answering query Q_2. Thus, we can use it to instrument the query to filter out irrelevant data early-on. For a query Q and sketch $\mathcal{P}$ we use $Q[\mathcal{P}]$ to denote the result of instrumenting Q to filter out data that does not belong to $\mathcal{P}$. Recall that $f_1 = [AL, DE]$. In $Q_2[\mathcal{P}_{state}]$ (see Fig. 1a), we apply a condition* `WHERE state BETWEEN 'AL'AND 'DE'` *to filter out data that does not belong to the sketch.*

By translating the sketch into a selection condition, we expose to the database system what data is relevant. Databases are already well-equipped to deal with such conditions and exploit existing physical design, e.g., use an index on `state`. However, sketches with a large number of fragments can result in conditions with a large number of disjunctions. We present optimizations to speed up such expressions.

Sketch Safety. So far we have assumed that if the provenance for a query is sufficient then so is the superset of the provenance encoded by a sketch. However, this is not always the case.

EXAMPLE 5. *Consider the partition of relation cities on attribute* `popden` *shown on the bottom of Fig. 1e. The first four tuples belong to the fragment for range g_2 since their population density is in* $[4001, 9000]$. *The remaining tuples belong to fragment g_1: $g_1 = \{t_5, t_6, t_7\}$ and $g_2 = \{t_1, t_2, t_3, t_4\}$. Only fragment g_2 contains tuples from the provenance of Q_2. Hence, the sketch corresponding to this partition is $\mathcal{P}_{popden} = \{g_2\}$. Evaluating Q_2 over g_2, we get* `(NY, 7000)` *which is different from the result for the full input. The reason is that g_2 contains only one tuple with state* `NY` *resulting in an average for this state that is higher than the one for* `CA`*.*

We call a sketch *safe* if evaluating the query over the data encoded by the sketch yields the same result as evaluating the query over the full input. We present a *sound* technique that determines safety statically, accessing only database statistics, but not the data.

Reusing Sketches for Parameterized Queries. Parameterized queries are used to avoid repeated optimization of queries that only differ in constants used in selections. Typically, an application uses a small number of parameterized queries, but executes many instances of each parameterized query. We develop a sound, but not complete, method that statically determines whether a sketch captured for one instance of a parameterized query can be used to answer another instance of this query. The problem of answering queries with sketches requires solving a problem that is closely related to query containment and, thus, unsurprisingly is undecidable in general. Thus, a sound method is the best we can hope for.

Provenance-based Data Skipping. We develop a framework for creating and using sketches that we refer to as *provenance-based data skipping* (PBDS). PBDS is used in a self-tuning fashion similar to automated materialized view selection: we decide when to create and when to use provenance sketches with the goal to optimize overall query performance. In this work, we assume read-only workloads which is common in OLAP and DISC systems. We leave maintenance of provenance sketches under updates for future work.

Contributions. Our main technical contributions are:

- We introduce provenance-based data skipping (PBDS), a novel method for analyzing at runtime what data is relevant for a query and introduce provenance sketches as a concise encoding of what subset of the input is relevant for a query.
- We develop techniques for capturing provenance sketches by instrumenting queries to propagate sketch annotations.
- We speed up queries by instrumenting them to filter data based on sketches. By exposing relevance data as selection conditions to the DBMS, existing physical design can be exploited.
- We present techniques for determining what sketches are safe for a query and for determining whether a sketch for an instance Q_1 of a parameterized query can be used to answer an instance Q_2.
- Using DBMS extensibility mechanisms, we implement PBDS using query instrumentation. We demonstrate experimentally that it leads to significant performance improvements. We compare PBDS with query answering with views (MVs) and show that these techniques are complementary. PBDS is more efficient than MVs for certain query types at significantly lower storage cost.

The remainder of this paper is organized as follows. We review related work in Sec. 2 and cover background in Sec. 3. Afterwards, we define provenance sketches in Sec. 4. Our approach for capturing and using sketches is discussed in Sec. 5 and 7. In Sec. 8, we present techniques for determining provenance sketch safety. We investigate how to resue sketches across queries in Sec. 9. Sec. 10 discussed a solution for integrating these techniques for self-tuning. We present experimental results in Sec. 11 and conclude in Sec. 12.

2 RELATED WORK

Our work is related to provenance capture techniques, compression and summarization of provenance, maintaining query results using provenance, and physical design and self-tuning techniques.

Physical Design and Self-tuning. Index structures [4, 7, 13, 17, 21, 44, 45, 52, 53, 62–64, 84, 84, 90], horizontal and vertical partitioning techniques [6, 18, 33, 56, 71, 77, 80, 87, 91], zone maps [26, 69], materialized views [3, 5, 8, 22, 43, 49, 50, 65], join indexes [13, 67, 76, 88], and many other physical design techniques have been studied intensively. However, databases fail to exploit physical design artifacts for important classes of selective queries such as top-k queries, because relevance of data cannot be determined statically for such queries. We close this gap by capturing relevance information at runtime and by translating it into selection conditions that DBMS optimizers are well-equipped to handle. Self-tuning techniques have a long tradition in databases [23, 33, 45]. Closely related to our work are automated selection of and query answering with materialized views (MVs) [2, 5, 22, 34, 43, 49, 50, 65, 78]. A technique similar to MVs in terms of trade-offs is re-use of data structures such as the hash table of a hash-aggregate [35]. In contrast to MVs, our technique has negligible storage requirements and can exploit existing physical design. Another disadvantage of MVs compared to sketches is that because of their larger size, they compete with the base data for buffer pool space. As we demonstrate in Sec. 11.8, reuse of MVs and sketches behaves quite differently, e.g., sketches may be reusable when a selection condition below an aggregation is modified which is not the case for MVs. However, MVs are sometimes superior for queries that filter the result of an aggregation. While sketches can also be used for such queries, a sketch may be less effective unless the view is large and/or the filter applied to its result is selective. Note that PBDS can also be combined with MVs. For instance, we can build a provenance sketch on top of a MV instead of on a base table which is beneficial for answering queries that use large MVs.

Provenance Capture. Many approaches for provenance capture, encode provenance as annotations on data and propagate such annotations through queries [46, 57, 72, 73]. Systems that capture database provenance include Perm [42], GProM [14], DBNotes [16], LogicBlox [46], Smoke [79], declarative Datalog debugging [59], ExSPAN [93], ProvSQL [85], Müller et al.'s approach [70], and Links [38]. In PBDS, we only have to generate a single sketch as the output and, thus, capture is significantly more efficient.

Compressing, Sketching, and Summarizing Provenance. Work on compressing and factorizing provenance such as [12, 20, 24, 68, 74, 75] avoids storing common substructures in the provenance more than once. Heinis et al. [51] studies how to trade space for computation when querying provenance graphs recursively. Malik et al. [68] use bloom filters to create compact sketches that over-approximate provenance. Chen et al. [24] apply similar ideas. Olteanu et al. [74, 75] study factorized representations of provenance and data. This line of work has lead to worst-case optimal algorithms for factorizing data wrt. to queries. Closely related are techniques for provenance summarization [9, 31, 41, 60, 61, 66], intervention-based methods for explaining aggregate query results [81, 82, 89] and other approaches for explaining outcomes [36, 37]. Some of these techniques use declarative descriptions of data, e.g., selection queries [36, 37, 61, 82]. Such summaries are typically not sufficient for our purpose, i.e., they may not encode a superset of the provenance. Our approach uses very compact (10s or 100s of bytes) declarative descriptions of a superset of the provenance which is sufficient for reproducing a query result.

Optimizing Operations with Provenance. Early work on provenance already recognized the potential of using provenance for optimizing performance. Pandas [54, 55] uses provenance to selectively update the outputs of of a workflow to reflect changes to the workflow's inputs. Provenance has been used to provision for answering what-if queries [15, 30–32] and to speed-up queries in interactive visualization [79]. Assadi et al. [15] create sketches over provenance to provision for approximate answering of what-if queries. In contrast to prior work which uses provenance (sketches) instead of the original input, our sketches act as a light-weight index that allows us to efficiently access relevant inputs.

3 BACKGROUND

In this section we introduce necessary background provenance and introduce relational algebra used in this work.

3.1 The Relational Data Model and Algebra

We use bold face (non-bold) to denote relation and database schemas (instances) respectively. The arity $arity(\mathbf{R})$ of $\mathbf{R}$ is the number of attributes in $\mathbf{R}$. Here we use bag semantics and for simplicity will sometimes assume a universal domain $\mathbb{U}$. That is, a relation R for schema $\mathbf{R}$ is a bag of tuples (elements of $\mathbb{U}^{arity(\mathbf{R})}$). We denote bags using $\{\!\!\{\cdot\}\!\!\}$ and use $t^n \in R$ to denote that tuple t appears with multiplicity n in relation R. Fig. 2 shows the bag relational algebra used in this work. Let $\text{SCH}(Q)$ denote the schema of query Q's result. We use $t.A$ to denote the projection of a tuple on a list of scalar expressions with renaming where $A = e_1 \rightarrow a_1, \ldots, e_n \rightarrow a_n$, each e_i is a scalar expression (an expression that returns a single value), and a_i is an attribute name. We use $\circ$ to denote concatenation of tuples. For convenience, we use $t^0 \in R$ to denote that the tuple t is not in R. The definitions of selection, projection, cross product, duplicate elimination, and set operations are standard. We use join $\bowtie$ as a shortcut for a crossproduct followed by a selection. Aggregation $\gamma_{f(a)\rightarrow b;G}(R)$ groups input tuples according to their values in attributes G and then computes the aggregation function f over the bag of values of attribute a for each group. Let $<_O$ denote a total order over the tuples of a relation R sorting on attributes O breaking ties using the remaining attributes. The top-k operator $\tau_{O,C}(R)$ returns the C smallest tuples from R wrt. $<_O$.

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

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

$$R \cup S = \{\!\!\{ t^{n+m} \mid t^n \in R \wedge t^m \in S \}\!\!\}$$

$$\gamma_{f(a) \rightarrow b;\, G}(R) = \{\!\!\{ g \circ f(R_g)^1 \mid g \in \text{GRPS}(R, G) \}\!\!\}$$

$$\text{GRPS}(R, G) = \{ t.G \mid t^n \in R \} \quad R_g = \{\!\!\{ (c)^n \mid n = \sum_{t^m \in R \wedge t.G = g \wedge t.a = c} m \}\!\!\}$$

$$\tau_{O,C}(R) = \{\!\!\{ t^n \mid t^m \in R \wedge n = max(0, min(m, C - pos(R, O, t))) \}\!\!\}$$

$$pos(R, O, t) = |\{\!\!\{ t_1^n \mid t_1^n \in R \wedge t_1 <_O t \}\!\!\}|$$

Figure 2: Bag Relational Algebra

3.2 Provenance and Sufficient Inputs

In the following, we are interested in finding subsets D' of an input database D that are sufficient for answering a query Q. That is, for which $Q(D') = Q(D)$. We refer to such subsets as sufficient inputs.

Definition 1 (Sufficient Input). *Given a query Q and database D, we call $D' \subseteq D$* sufficient *for Q wrt. D if*

$$Q(D) = Q(D')$$

Several provenance models for relational queries have been proposed in the literature [25]. Most of these models have been proven to be instances of the semiring provenance model [47, 48] and its extensions for difference/negation [39] and aggregation [11]. Our main interest in provenance is to determine a sufficient subset of the input database. Thus, even a simple model like Lineage which encodes provenance as a subset of the input database is expressive enough. We use $P(Q, D)$ to denote the provenance of a query Q over database D encoded as a bag of tuples and assume that $P(Q, D)$ is sufficient for Q wrt. D. For instance, we may construct $P(Q, D)$ as the union of the Lineage for all tuples $t \in Q(D)$. Our results hold for any provenance model that guarantees sufficiency.

4 PROVENANCE SKETCHES

We propose provenance sketches to concisely represent a superset of the provenance of a query Q (a sufficient subset of the input) based on horizontal partitions of relations. A sketch contains all fragments which contain at least one row from the provenance of Q. We limit the discussion to range-partitioning since it allows us to exploit existing index structures when using a sketch to skip data. However, note that most of the techniques we introduced are independent of the type of partitioning.

4.1 Range Partitioning

Given a set of intervals over the domains of a set of attributes $A \subset \mathbf{R}$, range partitioning determines membership of tuples in fragments based on which interval their values belong to. For simplicity, we define range partitioning for a single attribute a. Fig. 1e shows two range partitions for our running example.

Definition 2 (Range partition). *Consider a relation R and $a \in \mathbf{R}$. Let $\mathcal{D}(a)$ denote the domain of a. Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ be a set of intervals $[l, u] \subseteq \mathcal{D}(a)$ such that $\bigcup_{i=0}^{n} r_i = \mathcal{D}(a)$ 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:*

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

To determine ranges, we exploit the fact that most DBMS maintain statistics in the form of equi-depth histograms which provide us with a range-partitioning of a table on a column. However, our approach is compatible with any strategy for determining ranges.

A range partition of a relation divides the tuples of the relation into disjoint groups called *fragments*.

4.2 Provenance Sketches

Consider a database D, query Q, and a range partition $F_{\mathcal{R},a}$ of R. A provenance sketch $\mathcal{P}$ for Q according to $F_{\mathcal{R},a}$ is a subset of the ranges $\mathcal{R}$ of $F_{\mathcal{R},a}$ such that the fragments corresponding to the ranges in $\mathcal{P}$ fully cover Q's provenance within R, i.e., $P(Q, D) \cap R$. We use $\mathcal{R}(D, F_{\mathcal{R},a}(R), Q) \subseteq \mathcal{R}$ to denote the set of ranges whose fragment contains at least one tuple from $P(Q, D)$:

$$\mathcal{R}(D, F_{\mathcal{R},a}(R), Q) = \{ r \mid r \in \mathcal{R} \wedge \exists t \in P(Q, D) : t \in R_r \}$$

Definition 3 (Provenance Sketch). *Let Q be a query, D a database, R a relation accessed by Q, and $F_{\mathcal{R},a}(R)$ a range partition of R. We call a subset $\mathcal{P}$ of $\mathcal{R}$ a* ***provenance sketch*** *iff:*

$$\mathcal{P} \supseteq \mathcal{R}(D, F_{\mathcal{R},a}(R), Q)$$

We call a provenance sketch ***accurate*** *if*

$$\mathcal{P} = \mathcal{R}(D, F_{\mathcal{R},a}(R), Q)$$

We use $R_{\mathcal{P}}$, called the ***instance*** *of $\mathcal{P}$, to denote $\bigcup_{r \in \mathcal{P}} R_r$.*

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 Q (the instance $R_{\mathcal{P}}$ of $\mathcal{P}$). We call a sketch $\mathcal{P}$ *accurate* if it only contains ranges whose fragments contain provenance. We use $\mathcal{PS}$ to denote a set of provenance sketches for a subset of the relations in the database accessed by a query. Consider such a set $\mathcal{PS} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ where $\mathcal{P}_i$ is a sketch for relation R_i in database D and $R_i \neq R_j$ for $i \neq j$. We use $D_{\mathcal{PS}}$ to denote the database derived from database D by replacing each relation R_i for $i \in \{1, \ldots, n\}$ with $R_{\mathcal{P}_i}$. Note that we do not require that all relations of D are associated with a sketch. Abusing notation, we will use $D_{\mathcal{P}}$ to denote $D_{\{\mathcal{P}\}}$. Reconsider the running example in Fig. 1. Let $\mathcal{P}$ be the accurate provenance sketch of Q_2 using the range partition $F_{\mathcal{R},state}(cities)$. Recall that $P(Q_2, cities) = \{t_2, t_3\}$. Tuples t_2 and t_3 both belong to fragment f_1 since $CA \in [AL, DE]$. Thus, $\mathcal{P} = \{f_1\}$.

4.3 Sketch Safety

By construction we have $P(Q, D) \subseteq D_{\mathcal{PS}} \subseteq D$. Recall that $P(Q, D)$ is sufficient, i.e., $Q(P(Q, D)) = Q(D)$. However, as shown in Ex. 5 this does not guarantee that $Q(D_{\mathcal{PS}}) = Q(D)$, even when $\mathcal{PS}$ is accurate. We call a set of sketches **safe** for a query Q and database D if evaluating Q over the data described by the sketches returns the same result as evaluating it over D.

Definition 4 (Safety). *Let Q be a query, D be a database, and $\mathcal{PS}$ a set of sketches for Q. We call $\mathcal{PS}$* safe *for Q and D iff:*

$$Q(D_{\mathcal{PS}}) = Q(D)$$

Obviously, only safe sketches are of interest. In Sec. 8 we present a method for testing which attributes are safe for building sketches for a query. For that we define attributes to be safe for a database and query Q, if any sketches created over these attributes are safe.

Definition 5 (Attribute Safety). *Let D be a database, Q a query, and A a set of attributes from the schema of a relation R accessed by Q. We call A* safe *for Q and D if for every range partition $F_{\mathcal{R},A}$ of R, all sketch $\mathcal{P}$ based on $F_{\mathcal{R},A}$ are safe for Q and D. A set of attributes $X = \bigcup_1^n X_i$ where each X_i belongs to a relation R_i accessed by Q is safe for D and Q, if each X_i is safe for D and Q.*

5 PROVENANCE SKETCH CAPTURE

We now discuss how to capture provenance sketches through query instrumentation. We first review how queries are instrumented to propagate provenance using Lineage [25, 27, 48] where the provenance of a query result is the set of input tuples that were used to derive the result. Most approaches operate in two phases: 1) **annotate** each input tuple with a singleton set containing its identifier, e.g., the ones shown to the left of each tuple in Fig. 1b and 2) **propagate** these annotations through each operator of a query such that each (intermediate) query result is annotated with its provenance.

Example 6. *To capture the Lineage of each result tuple of the query Q_2 from Ex. 3 we annotate each tuple t_i from the cities table (see Fig. 1b) with a singleton set $\{t_i\}$. Then annotations are propagated through the operators of Q_2. At last we get one result tuple with annotation $\{t_2, t_3\}$ (see Fig. 1d). The annotation $\{t_2, t_3\}$ means that the result tuple (CA, 5500) of Q_2 was produced by combining input tuples (6000, San Diego, CA) and (5000, Sacramento, CA).*

Our approach for computing sketches also operates in two phases. However, we annotate each tuple with the fragment the tuple belongs to instead of the tuple identifier. For a partition F, the size of the annotation is determined by $|F|$, i.e., the number of fragments of F. Since the partitions are fixed for a query, the annotations used for capture only need to record which fragments are present (for each partition we are using). This can be done compactly using bit sets. A partition with n fragments is encoded as a vector of n bits. We refer to this as the *bitset encoding* of a sketch. For instance, for the range-partition on attribute `state` from Fig. 1e, the fragments f_1 and f_3 are encoded as `1000` and `0010`. Assume we range-partition relation cities on attribute `state` using the intervals shown in Fig. 1e. Each input is annotated with the singleton fragment it belongs to ($\{f_1\}$, $\{f_3\}$ and $\{f_4\}$) as shown in Fig. 1b. Then, the result tuple of Q_2 (Fig. 1d) is generated based on input tuples $\{t_2, t_3\}$ which are both annotated with $\{f_1\}$. Thus, this tuple is annotated with $\{f_1\}$.

5.1 Initializing Annotations

We now discuss how to seed the tuple annotations for a query Q according to a set of range partitions $\mathcal{F} = \{F_1, \ldots, F_m\}$ over database D. Let F_i be the partition for relation R_i where $i \in [1, m]$. To simplify the presentation, we assume that no relation is accessed more than once by Q, but our approach also handles multiple accesses. Furthermore, we assume that we build sketches on all relations accessed by the query (we may want to omit relations for which the sketch would not be selective). Recall that a range partition (Def. 2) assigns tuples to fragments based on their value in an attribute a and a set of ranges over the domain of a. We add a projection on top of R_i to compute and store each row's fragment in a column λ_{R_i} computed using a **CASE** expression. We use $\textsc{Init}_{F_i}(R_i)$ to denote this instrumentation step. In relational algebra, we use $\mathsf{select}(\theta_1 \mapsto e_1, \ldots, \theta_n \mapsto e_n)$ to denote an expression that returns the result of the first e_i for which condition θ_i evaluates to true and returns null if all θ_i fail. We use $\textsc{sng}(i)$ to denote the singleton bit set for $\{f_i\}$. For a range partition $F_{\mathcal{R},a}(R)$ with ranges $\mathcal{R} = \{r_1, \ldots, r_n\}$ we generate the query:

$$\textsc{Init}_F(R) \coloneqq \Pi_{\mathrm{R},\mathsf{select}(a \in r_1 \mapsto \textsc{sng}(1), \ldots, a \in r_n \mapsto \textsc{sng}(n))}(R) \tag{1}$$

$$\textsc{Prop}(\mathcal{F}, R) = \textsc{Init}_{\mathcal{F}}(R) \tag{r_0}$$

$$\textsc{Prop}(\mathcal{F}, \Pi_A(Q)) = \Pi_{A,\Lambda}(\textsc{Prop}(\mathcal{F}, Q)) \tag{r_1}$$

$$\textsc{Prop}(\mathcal{F}, \sigma_\theta(Q)) = \sigma_\theta(\textsc{Prop}(\mathcal{F}, Q)) \tag{r_2}$$

$$\textsc{Prop}(\mathcal{F}, \gamma_{f(a);G}(Q)) = \begin{cases} \Pi_{a,G,\Lambda}(\gamma_{f(a);G}(Q) \bowtie_{f(a)=a \wedge G=G} \textsc{Prop}(\mathcal{F}, Q)) & \text{if } f = min \vee f = max \\ \gamma_{f(a),bitor(\Lambda);G}(\textsc{Prop}(\mathcal{F}, Q)) & \text{otherwise} \end{cases} \tag{r_3}$$

$$\textsc{Prop}(\mathcal{F}, Q_1 \times Q_2) = \textsc{Prop}(\mathcal{F}, Q_1) \times \textsc{Prop}(\mathcal{F}, Q_2) \tag{r_4}$$

$$\textsc{Prop}(\mathcal{F}, \tau_{O,C}(Q)) = \tau_{O,C}(\textsc{Prop}(\mathcal{F}, Q)) \tag{r_5}$$

$$\textsc{Prop}(\mathcal{F}, Q_1 \cup Q_2) = \textsc{Prop}(\mathcal{F}, Q_1) \cup \textsc{Prop}(\mathcal{F}, Q_2) \tag{r_6}$$

$$\textsc{Instr}(\mathcal{F}, Q) = \gamma_{bitor(\Lambda) \rightarrow \Lambda}(\textsc{Prop}(\mathcal{F}, Q)) \tag{r_7}$$

Figure 3: Instrumentation rules for sketch capture

For example, to instrument the relation access `cities` (Fig. 1b) from query Q_2 using the partition F_{state} (Fig. 1e), we generate query Q_{INIT} shown below (written in SQL for legibility). Based on the value of attribute `state` we assign tuples to fragments of F_{state}.

```
SELECT popden, city, state,
  CASE  WHEN  state >= 'AL' AND state <= 'DE' THEN '1000'
        WHEN  state >= 'FL' AND state <= 'MI' THEN '0100'
        WHEN  state >= 'MN' AND state <= 'OK' THEN '0010'
        WHEN  state >= 'OR' AND state <= 'WY' THEN '0001'
    END AS λ_state
FROM cities
```

QINIT

5.2 Propagating Annotations

We now discuss how to instrument a query to propagate annotations to generate a single output tuple storing the sketch(es) for the query. We denote the set of attributes of an instrumented query storing provenance sketches as Λ. Given a set of range partitions $\mathcal{F}$ over database D and query Q, we use $\textsc{Instr}(\mathcal{F}, Q)$ to denote the result of instrumenting the query to capture a sketch for $\mathcal{F}$. For two lists of attributes $A = (a_1 \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ we write $A = B$ as a shortcut for $\bigwedge_{i \in \{1,\ldots,n\}} a_i = b_i$. We apply similar notation for bulk renaming $A \rightarrow B$ and function application, e.g., $f(A)$ denotes $f(a_1), \ldots, f(a_n)$. We assume the existence of an aggregation function $\textsc{bitor}$ which computes the bit-wise OR of a set of bitsets. For example, in Postgres this function exists under the name `bit_or`. The rules defining $\textsc{Instr}(\cdot)$ are shown in Fig. 3. As the last step of the rewritten query $\textsc{Instr}(\cdot)$ we apply $\textsc{bitor}$ aggregation to merge the sketch annotations of the results of the query (r_7). The input to this aggregation is generated using $\textsc{Prop}(\mathcal{F}, Q)$ which recursively replaces operators in Q with an instrumented version.

Rule r_0 initializes the sketch annotations for relation R using $\textsc{Init}(\cdot)$ as introduced in Sec. 5.1. For projection we only need to add the Λ columns from its input to the result schema (r_1). Selection is applied unmodified to the instrumented input (r_2). A result tuple of an aggregation operator with group-by is produced by evaluating the aggregation function(s) over all tuples from the group. Thus, if each tuple t from a group is annotated with a set of fragments that is sufficient to produce t, then the union of these fragments is sufficient for reproducing the result for this group. Since aggregation is non-monotone, this is only correct if we use a partitioning that has been determined to be *safe* for the query using the method from Sec. 8. Hence, we union the provenance sketches for each group using the bitwise or aggregation function *bitor* (r_3), e.g., `1000` and `0010` will be merged producing `1010`. For aggregation functions *min* and *max* it is sufficient to only include tuples with the min/max value in attribute a. We implement this by selecting a single tuple with the min/max value for each group. For cross product we compute the cross product of the instrumented inputs (r_4). For the top-k operator we apply the operator to its instrumented input (r_5). For union we union the instrumented inputs (r_6).

THEOREM 1. *Consider a query Q, database D, and a set of range-partitions $\mathcal{F}$ for attributes that are safe for Q and D. Then* Instr$(\mathcal{F}, Q)(D)$ *produces a safe sketch.*

PROOF. We prove this by induction over the relational algebra expression. Base case (r_0 [table access]): Since INIT$_F(R)$ annotates each tuple t in R with the fragment f it belongs to, i.e., $t \in f$, the sketch produced by INSTR$(Q, \mathcal{F})$ contains all tuples from R and, thus, is trivially safe. Inductive Step: assume given a query Q' and database D, if exists tuple t' such that $t' \in Q'(D)$, then $t' \in Q'(D_{\mathcal{P}_{t'}})$. Next, We prove this is also hold for the query Q where $Q = op(Q')$ after applying our instrumentation rules for this query operator (op). In the following, we prove each op individually. r_1 [Π_A]: here A represents the set of attributes used in the projection. Assume $t \in \Pi_A(Q'(D))$, then we have $t = t'.A$ and $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'}}$, thus $t' \in Q'(D_{\mathcal{P}_t})$. Since $t = t'.A$, we get $t \in \Pi_A(Q'(D_{\mathcal{P}_t}))$. r_2 [σ_θ]: Assume $t \in \sigma_\theta(Q'(D))$, then we have $t = t'$ and $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'}}$, similarly, $t \in \sigma_\theta(Q'(D_{\mathcal{P}_t}))$. r_3 [γ]: assume $t \in \gamma(Q'(D))$ and t is derived from $t'_1, t'_2, ..., t'_n$ where $t'_i \in Q'(D)$. Then $t = \gamma(\{t'_1, t'_2, ..., t'_n\})$ and $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'_1, t'_2, ..., t'_n}} = D_{\mathcal{P}_{t'_1}} \cup D_{\mathcal{P}_{t'_n}} \cup ... \cup D_{\mathcal{P}_{t'_n}}$. Since $\{t'_1, t'_2, ..., t'_n\} \in Q'(D_{\mathcal{P}_{t'_1}}) \cup Q'(D_{\mathcal{P}_{t'_n}}) \cup ... \cup Q'(D_{\mathcal{P}_{t'_n}}) = Q'(D_{\mathcal{P}_t})$, we get $t \in \gamma(Q'(D_{\mathcal{P}_t}))$. When $\gamma \in \{\min, \max\}$, only one t' in $\{t'_1, t'_2, ..., t'_n\}$ is enough to derive t, the proof is same. r_4 [$Q'_L(D) \times Q'_R(D)$]: assume $t'_L \in Q'_L(D)$ and $t'_R \in Q'_R(D)$, then $t = t'_L \times t'_R$ and $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'_L}} \cup D_{\mathcal{P}_{t'_R}}$, thus $t'_L \times t'_R \in Q'_L(D_{\mathcal{P}_{t'_L}}) \times Q'_R(D_{\mathcal{P}_{t'_R}}) = Q'_L(D_{\mathcal{P}_t}) \times Q'_R(D_{\mathcal{P}_t})$, that is, $t \in Q'_L(D_{\mathcal{P}_t}) \times Q'_R(D_{\mathcal{P}_t})$. r_5 [$\tau_{O,C}(D)$]: Assume $t \in \tau_{O,C}(Q'(D))$, then we have $t = t'$ and $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'}}$, similarly, $t \in \tau_{O,C}(Q'(D_{\mathcal{P}_t}))$. r_6 [$Q'_L(D) \cup Q'_R(D)$]: assume $t \in Q'_L(D) \cup Q'_R(D)$ and $t = t'$ where $t' \in Q'_L(D)$, then $D_{\mathcal{P}_t} = D_{\mathcal{P}_{t'}}$ such that $t' \in Q'_L(D_{\mathcal{P}_t}) \in Q'_L(D_{\mathcal{P}_t}) \cup Q'_R(D_{\mathcal{P}_t})$. Thus, $t \in Q'_L(D_{\mathcal{P}_t}) \cup Q'_R(D_{\mathcal{P}_t})$. □

EXAMPLE 7. *Reconsider query Q_2 from our running example. We now explain the steps of generating* Instr(Q_2, F_{state}). *Fig. 5 shows the resulting query and Fig. 4 shows its intermediate and final result(s). In relational algebra, query Q_2 can be written as $Q_{total} := \tau_{desc,1}(\Pi_{state, sd \cdot -1 \rightarrow desc}(\gamma_{state; avg(popden) \rightarrow sd}(cities)))$. Since we defined top-k operator order by* **ASC**, *we represents the* **DESC** *by multiplying sd with -1. We start from the relation access which is instrumented as shown at the end of Sec. 5.1 (④ in Fig. 5). We apply r_3 to instrument the aggregation to propagate the values in $\lambda_{F_{state}}$. The rule uses* bitor *to aggregate input sketches for each group (see ③ in Fig. 5 and Fig. 4c). Since in SQL code, the* **ORDER BY** `avgden` **DESC LIMIT** `1` *was represented by a projection and top-k together, i.e., $\tau_{desc,1}(\Pi_{state, sd \cdot -1 \rightarrow desc})$, we apply the rules r_1 and r_5, the corresponding rewrite at the SQL level does not require any modification (② in Fig. 5 and Fig. 4d). Using r_7, we then apply* bitor *aggregation to generate the final sketch: $\{f_1\}$ (① in Fig. 5 and Fig. 4b).*

popden	city	state	$\lambda_{F_{state}}$
4200	Anchorage	AK	1000
6000	San Diego	CA	1000
5000	Sacramento	CA	1000
7000	New York	NY	0010
2000	Buffalo	NY	0010
3700	Austin	TX	0001
2500	Houston	TX	0001

(a) Result of ④

$\lambda_{F_{state}}$
1000

(b) Result of ①

state	avgden	$\lambda_{F_{state}}$
CA	5500	1000
NY	4500	0010
AK	4200	1000
TX	3100	0001

(c) Result of ③

state	avgden	$\lambda_{F_{state}}$
CA	5500	1000

(d) Result of ②

Figure 4: Intermediate and final result(s) of INSTR(F_{state}, Q_2)

```
①  SELECT bitor(λFstate) AS λFstate                  (r7)
   FROM (
     SELECT state,
③           avg(popden) AS avgden,                   (r3)
            bitor(λFstate) AS λFstate
     FROM (
④              QINIT   (see Sec. 5.1)                (r0)
③           ) p1                                     (r3)
     GROUP BY state
②    ORDER BY avgden DESC                         (r1) (r5)
     LIMIT 1) p2
```

Figure 5: Instrumented Query INSTR(F_{state}, Q_2)

5.3 Optimizations

Our instrumentation rules preserve the structure of the input query in most cases. Thus, the majority of overhead introduced by instrumentation is based on evaluating 1) **CASE** expressions and 2) bitor aggregations. For 1) to initialize a sketch with n fragments, we can apply binary search to test the membership of a value v in range r_i which reduces the runtime from $O(n)$ to $O(\log n)$. We implemented this optimization as UDFs written in C in MonetDB and Postgres, two systems we use in our experimental evaluation. For 2) if n is large, then singleton sets of fragments can be encoded more compactly by storing and propagating the position of the single bit set as a fixed-size integer value instead of storing and propagating a full bitset. This encoding can be retained until we encounter an aggregation and need to union bitsets which we call *delay* method. Furthermore, in Postgres, this bitor aggregation function results in unnecessary creation of $n-1$ new bitsets when calculating the bitwise or of n bit sets. Also, bitwise or is applied one byte at a time. We improve this implementation by computing the operation one machine-word at a time and by avoiding unnecessary creation of intermediate bitsets which we call *No-copy* method.For MonetDB we implement BITOR as a user-defined aggregation function in C.

6 ATTRIBUTE AND PARTITION SELECTION

Selecting Attributes. An important decision when creating a sketch is what attribute(s) A the sketch will be created on. The choice of attributes can significantly affect the size of the sketch instance. The most important factors are (i) does A have a sufficiently large number of distinct values to support fine-grained sketches, (ii) does the existing physical design support skipping of data based on sketch build on A, and (iii) are the values of A correlated with the query's provenance, i.e., how predictive is a tuple's A-value(s) of the tuple belonging to query's provenance. Primary key (PK) attributes typically fare well for (i) and (ii), possibly at the cost of being suboptimal wrt. (iii).

Range Partition Selection. Most DBMS maintain statistics in the form of equi-depth histograms which provide us with a range-partitioning of a table on a column. However, our approach is compatible with any strategy for determining ranges. If no histogram of with a sufficiently large number of buckets exists, then we instruct the DBMS to build a new histogram. Based on our experimental result, we recommend 10,000 fragment sketches as a solid choice that provides the best trade-off between capture and use performance for most datasets and workloads (testing on datasets between 1GB to more than 100GB in size). If the number of distinct values of a column is less than 10,000, we place each value in a separate range.

7 USING PROVENANCE SKETCHES

Once a sketch $\mathcal{P}$ has been captured, we can utilize it to speed up the subsequent execution of queries. For that we have to instrument the query to filter out data that does not belong to the sketch. This is achieved by decoding the sketches into selection conditions and applying these conditions in selection operators on top of every relation access that is covered by a sketch. Recall that we use $Q[\mathcal{P}]$ to denote the result of instrumenting query Q using sketch $\mathcal{P}$. $Q[\mathcal{P}]$ is defined as the identity function on all operators except for table access operators. Let F be a range-based partition of a relation R on attribute a using ranges $\mathcal{R} = (r_1, \ldots, r_n)$ and $\mathcal{P} = \{f_{i_1}, \ldots, f_{i_m}\}$ be a sketch based on F. We generate a condition $\bigvee_{j=1}^{m} a \in r_{i_j}$ to filter R based on F. Thus, the instrumentation rule for applying the sketch to R is:

$$R[\mathcal{P}] \coloneqq \sigma_{\bigvee_{j=1}^{m} a \in r_{i_j}}(R) \tag{2}$$

For example, the query Q_2 in the running example would be rewritten to $Q_2[\mathcal{P}_{state}]$ (See Fig. 1a).

Instrumentation for other partition schemes operates in a similar fashion, e.g., for a hash-based partition we produce a disjunction $\bigvee_{j=1}^{m} h(a) = i_j$.

7.1 Optimizations

Databases can exploit physical design to evaluate the type of selection conditions we create for range-based sketches. However, if $|\mathcal{P}|$ is large, i.e., the sketch contains a large number of fragments, then the size of the selection condition that has to be evaluated may outweigh this benefit. Furthermore, if the database has to resort to a full table scan, then we pay the overhead of evaluating a condition that is linear in $|F|$ for each tuple. We now discuss how to improve this by reducing the number of conditions and/or improving the performance of evaluating these conditions. First off, if a sketch contains a sequence of adjacent fragments $f_i, \ldots, f_j$ for $i < j$, we can replace the conditions $\bigvee_{k=i}^{j} a \in r_k$ with a single condition $a \in \bigcup_{k=i}^{j} r_k$. Reconsider the sketch $\mathcal{P} = \{f_1, f_2\}$ from the example above. Since these two fragments are adjacent, we can generate a single condition $state \in [AL, MI]$ instead of $state \in [Al, DE] \lor state \in [FL, MI]$. Note that the condition generated for a range partition checks whether an attribute value is an element of one of the ranges corresponding to the fragments of the sketch. Since these ranges are ordered, we can apply binary search to improve the performance of evaluating a condition with n disjunctions from $O(n)$ to $O(\log n)$. We implemented a Postgres extension to be able to exploit zone maps (brin indexes in Postgres) to skip data based on such a condition.

8 TESTING SKETCH SAFETY

In this section, we develop a sound method that determines whether a given set of attributes X is safe for a query Q and database D. Since we want to determine upfront whether the sketches on a set of attributes X are safe before paying the cost of creating such sketches, instead of fully accessing D, we design a method which only accesses Q and basic statistics SD of D, specifically the minimum and maximum values of each column. Given a set of attributes X, query Q and statistics of D SD as input, this algorithm constructs a universally quantified logical formula without free variables such that if this formula evaluates to true, then X is safe for Q and D. Similar to recent work on query equivalence checking [92], we utilize an SMT solver [28] to check whether the formula is true by rewriting it into negated existential form (a universally quantified formula is true if its negation is unsatisfiable). For example, to test $\forall a : a < 10$, we check whether $a \geq 10$ is unsatisfiable. The formula we construct can be evaluated by SMT solvers such as Z3 [29] as long as conditions, projection expressions, and aggregation functions only utilize operations and comparisons that are supported by the SMT solver.

Before presenting our approach we first state a negative result motivating the decision to develop an algorithm that is only sound, but not complete.

THEOREM 2. *There cannot exist a sound and complete algorithm that determines safety of a set of attributes for a query Q and D without accessing D.*

PROOF. We prove this theorem by demonstrating that there exists an attribute a, two databases D and D', and a query Q such that an accurate provenance sketch created for Q according to some range partitioning of a over D is safe while the sketch created for D' is unsafe. Since the algorithm is not allowed to inspect the database it cannot distinguish between D and D' and, thus cannot determine whether the attribute is safe for Q for a given database. Consider query Q_2 from Fig. 1a and the range partition F_{popden}. As discussed in Sec. 1, the sketch $\mathcal{P}_{popden}$ created based on this partition is unsafe for Q_2 and the instance of the cities relation shown in Fig. 1b. However, consider the database D' that only consists of tuples t_2 and t_3 from Fig. 1b. The provenance sketch for Q_2 using F_{popden} contains the fragment f corresponding to range g_2 (all tuples with a population density between 4001 and 9000). Since $f = D'$, we have $Q(f) = Q(D')$. □

Rationale and Considerations. Before explaining our safety checking technique in more detail, we first provide some rationale for its design and an intuition for what attributes are safe for which classes of queries. Any set of attributes is safe for monotone queries (for which $D \subseteq D' \Rightarrow Q(D) \subseteq Q(D')$).

Definition 6 (Monotone Query). *A query Q is* monotone *iff:*

$$\forall D, D' : D \subseteq D', Q(D) \subseteq Q(D')$$

One important result regarding the safety of attributes is that for monotone queries any set of attributes is safe. This result follows immediately from the definition of monotonicity, the sufficiency of provenance, and the fact that sketches overestimate provenance.

Theorem 3 (All Attributes are Safe for Monotone Queries). *Given a monotone query Q and database D, any set of attributes X is safe for Q and D.*

Proof. Consider a set of provenance sketches $\mathcal{PS}$ on X. We have

$$P(Q, D) \subseteq D_{\mathcal{PS}} \subseteq D$$

Because Q is monotone, $D \subseteq D'$ implies $Q(D) \subseteq Q(D')$. Thus, we get

$$Q(P(Q, D)) \subseteq Q(D_{\mathcal{PS}}) \subseteq Q(D) \tag{3}$$

Recall that the provenance $P(Q, D)$ is sufficient for Q and D, i.e.,

$$Q(P(Q, D)) = Q(D) \tag{4}$$

Equation (3) and Equation (4) imply that $Q(D) = Q(D_{\mathcal{PS}})$. □

For queries involving aggregation, a major challenge stems from the fact that provenance sketches encode a superset of the provenance. Thus, they may contain a subset of the input tuples for a group whose result may not contribute to any query result tuple. This can lead to the aggregation producing a different aggregation function result for such a group which in turn may lead to a different final query result. We already showcased this problem in Ex. 5. This problem can be avoided by creating the sketch on a subset of the group-by attributes, i.e., the group-by attributes of an aggregation query are safe. Non-group-by attributes are safe when the results produced for partial groups will not affect the final query result. Thus, our safety check procedure needs to reason about how the values of a tuple in $Q(D)$ are related to values of the corresponding tuple in $Q(D_{\mathcal{PS}})$. For instance, for aggregation function `count`, the count of a partial group included in a sketch is guaranteed to be smaller than the count for the full group. Then, for a query that returns the top-k counts or uses a **`HAVING`** condition which checks that the count is larger than a threshold, groups that did not make the cut in the evaluation of Q over D will also not be in the result when only the partial group included the sketch is used. Thus, for such queries, also non-group-by attributes are safe. For instance, if we would change the aggregation function in query Q_2 from Fig. 1 to be `count`, then the sketch on `popden` would be safe.

8.1 Generalized Containment

Our approach utilizes a generalization of the subset relationship between two relations to be able to express that, e.g., a count aggregation returns a subset of the groups over the sketches, but the counts produced by $Q(D_{\mathcal{PS}})$ (running the query over the sketches) are smaller than the counts for $Q(D)$. Consider following example:

Example 8. *Reconsider Fig. 1 and let Q_{total} be query Q_2 where the aggregation function is replaced with* `sum(popden)` **`AS`** `sd`*. Then this query returns* `(CA, 11000)` *and the provenance of Q_{total} is $\{t_2, t_3\}$. Consider creating a sketch PS_{total} on the partition F_{popden} shown in Fig. 1e, all cities in the provenance belong to g_2 ([4001,9000]). Because this fragment contains row t_4 (New York), evaluating the aggregation subquery (noted as Q_{agg}) over the sketch returns a smaller result for NY (7000). However, this does not affect the final result, because CA already had a larger sum for the full group. Note that this does not just work out for this particular example instance. Since population density is positive, the sum for any partial group included in the sketch will be smaller than for $Q(D)$ and, thus, these groups will be filtered out by the top-1 operator.*

The definition of *generalized containment* shown below allows us to express such complex relationships where one relation contains some tuples that also exist in another relation, albeit with different attribute values that obey some constraints.

Definition 7 (Generalized Containment). *Let $R(a_1, \ldots, a_n)$ and $R'(b_1, \ldots, b_n)$ be two relations with the same arity. Furthermore, let Ψ be a boolean formula over comparisons of the form $a_i \diamond b_i$ where $i \in [1, n]$ and $\diamond \in \{\leq, =, \geq\}$. The generalized containment relationship $R \precsim_{\Psi} R'$ based on Ψ holds for R and R' if there exists a mapping $\mathcal{M} \subseteq R \times R'$ that fulfills all of the following conditions:*

$$\forall t \in R : \exists t' \in R' : \mathcal{M}(t, t')$$

$$\forall t_1, t_2, t_1', t_2' : \mathcal{M}(t_1, t_1') \land \mathcal{M}(t_2, t_2') \land (t_1 = t_2 \lor t_1' = t_2') \rightarrow t_1 = t_2 \land t_1' = t_2'$$

$$\forall (t, t') \in \mathcal{M} : (t, t') \models \Psi$$

The first and second conditions ensure that every tuple from R is "matched" to exactly one tuple from R'. The third condition ensures that all pairs of matched tuples fulfill condition Ψ. Note that $R \subseteq R'$ is a special case of generalized containment where $\Psi = \bigwedge_{i=1}^{n} a_i = b_i$. In the following, we will use generalized containment to model the relationship between (intermediate) results of a query over the full input database and over the instance of a set of provenance sketches. In this scenario, the two relations we are comparing have the same schema. To avoid ambiguities in Ψ, for each attribute a in the schema, we use $a^{\mathcal{PS}}$ to refer to the corresponding attribute over the instance of the sketches. Reconsider Ex. 8, the relationship between the results of the subquery Q_{agg} over D and $D_{\mathcal{P}}$ can be encoded as the generalized containment relationship $Q_{agg}(D_{\mathcal{P}}) \precsim_{sd^{\mathcal{PS}} \leq sd \land state^{\mathcal{PS}} = state} Q_{agg}(D)$.

8.2 Inference Rules

Given a query Q, a set of attributes X from the database D and statistics SD of D, we construct a logical formula $gc(Q, X, SD)$ which takes Q, X, and SD as input and returns true iff $Q(D_{\mathcal{PS}})$ is generalized contained in $Q(D)$ according to a formula $\Psi_{Q,X,SD}$ for any set of sketches $\mathcal{PS}$ created on X for D. For instance, for an aggregation, $\Psi_{Q,X,SD}$ encodes how the aggregation function results for D and $D_{\mathcal{PS}}$ are related to each other. Intuitively, $gc(Q, X, SD)$ does encode constraints that have to hold for attribute values of any tuple produced by $Q(D_{\mathcal{PS}})$ and/or by $Q(D)$. For instance, if the query contains a selection on a condition $a < 10$ then all result tuples of the selection are guaranteed to fulfill $a < 10$. At last, we demonstrate that this type of generalized containment (based on $gc(Q, X, SD)$) does imply $Q(D_{\mathcal{PS}}) = Q(D)$. In the following, we first introduce some auxiliary constructs which are used to define $gc(Q, X, SD)$ and then discuss the rules that define gc (Fig. 6). For simplicity of exposition we assume that attribute names are unique.

pred(Q). We use *pred* to record conditions which have to be fulfilled by all tuples produced by query Q and its subqueries. *pred* is defined recursively as shown below. For instance, selection and join conditions are added to *pred*, since all tuples produced by such operators have to fulfill these conditions. As an example consider the query $Q := \sigma_{a=5}(\Pi_a(\sigma_{b<4}(R)))$, we get $pred(Q) = (a = 5 \wedge b < 4)$. Note that we are using database statistics SD to bound the values of tuples from input relation R. $min(a)$ ($max(a)$) denotes the smallest (largest) value in attribute a.

$$pred(Q) = \begin{cases} \bigwedge_{a \in SCH(R)} a \geq min(a) \wedge a \leq max(a) & \textbf{if } Q \text{ is a relation} \\ pred(Q_1) \wedge pred(Q_2) & \textbf{if } Q = Q_1 \times Q_2 \\ pred(Q_1) \wedge pred(Q_2) \wedge \theta & \textbf{if } Q = Q_1 \bowtie_\theta Q_2 \\ pred(Q_1) \wedge \theta & \textbf{if } Q = \sigma_\theta(Q_1) \\ pred(Q_1) \vee pred(Q_2) & \textbf{if } Q = Q_1 \cup Q_2 \\ pred(Q_1) & \textbf{otherwise} \end{cases}$$

expr(Q). This formula encodes relationships between values of attributes in the result of the query and its subqueries. For every generalized projection, we record how the value of attributes in the output of the projection are related to the values of attributes in its input. For example, for $Q := \Pi_{a+b \to x, c+d \to y}$, we get $expr(Q) = (a + b = x \wedge c + d = y)$.

$$expr(Q) = \begin{cases} \emptyset & \textbf{if } Q \text{ is a relation} \\ expr(Q_1) \wedge expr(Q_2) & \textbf{if } Q = Q_1 \times Q \textbf{ or } Q_1 \bowtie_\theta Q_2 \\ expr(Q_1) \wedge \bigwedge_{i=1}^{n} e_i = b_i & \textbf{if } Q = \Pi_{e_1 \to b_1, \ldots, e_n \to b_n}(Q_1) \\ expr(Q_1) \vee expr(Q_2) & \textbf{if } Q = Q_1 \cup Q_2 \\ expr(Q_1) & \textbf{otherwise} \end{cases}$$

We use $conds(Q)$ to denote $pred(Q) \wedge expr(Q)$. We will show that the generalized containment $Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$ holds if $gc(Q, X, SD)$ is valid. Importantly, we will then prove that this implies that $Q(D_{\mathcal{PS}}) = Q(D)$. We define $gc(Q, X, SD)$ using a set of rules, one for each operator of our algebra. We apply these rules recursively in a bottom-up traversal. That is, whether $gc(Q, X, SD)$ holds is based on the root operator of Q and whether gc holds for the root's children $gc(Q_1, X, SD)$ and some additional conditions. The rules are shown in Fig. 6. We use ATTRS(Q) to denote the set of attributes of the relations accessed by Q. Furthermore, for a subquery Q_1 (Q_2) we use X_1 (X_2) to denote the subset of X contained in relations accessed by Q_1 (Q_2). For any subquery Q that does not contain of attributes for which we want to test safety ($X = \emptyset$), we know that $D_{\mathcal{PS}}$ contains the original relations from D. Thus, $Q(D_{\mathcal{PS}}) = Q(D)$ for any such subquery and we set $\Psi_{Q,X,SD}$ to the equality on all attributes and $gc(Q, X, SD) = \textbf{true}$. The $gc(Q, X, SD)$ condition for most operators requires that gc holds for the operator's input. Additionally, operator-specific conditions apply.

Table Access. For a table access operator we know that $R_{\mathcal{PS}} \subseteq R$. Thus, we set $gc(R, X, SD) = \textbf{true}$ and $\Psi_{R,X,SD}$ to the equality on all attributes, i.e., $\Psi_{R,X,SD} = \bigwedge_{a \in \text{SCH}(Q)} a^{\mathcal{PS}} = a$.

Selection. We check whether the condition $\theta^{\mathcal{PS}}$ evaluated on any tuple from $D_{\mathcal{PS}}$ has to imply that the condition holds for the corresponding tuple over D (θ) (We use D (θ) to represent the evaluation of θ over D), because that implies generalized containment. Since no additional attributes are created by these operators, thus $\Psi_{Q,X}$ is the same as Ψ_{Q_1,X_1}.

Aggregation. We check whether the conditions for the input of the aggregation (Q_1) do imply that all group-by attributes are equal on D and $D_{\mathcal{PS}}$. Here, we use X_1 to represent the attributes in X which are from relations accessed by Q_1 (for aggregation we have $X_1 = X$). If generalized containment holds Q_1 and the group-by attributes are equal for all inputs, then generalized containment will hold for the result. To determine $\Psi_{Q,X,SD}$ which, in addition to constraints on the attributes from Q_1, encodes how the aggregation function result (attributes b and $b^{\mathcal{PS}}$) for a group over D and $D_{\mathcal{PS}}$ are related to each other, we have to consider several cases. (i) if X_1 is a subset of (equivalently guaranteed to be equal to) the group-by attributes, if then calculating the aggregation function over the sketch instance yields the same result as over the full database, because each group is contained in exactly one fragment of the partition on which the sketch is build on. Thus, either all or none of the tuples of a group are included in $D_{\mathcal{PS}}$ and for all groups included in $D_{\mathcal{PS}}$, the aggregation function result will be the same in $Q(D_{\mathcal{PS}})$ and $Q(D)$; (ii) for aggregation functions that are monotone (e.g., count, max, or sum over positive numbers) we know that the aggregation function result produced for a group that occurs in $Q(D_{\mathcal{PS}})$ has to be smaller than or equal to the result for the same group in $Q(D)$. Thus, if the constraints we have derived for the input of the aggregation imply that the input attribute a for the aggregation function is larger than 0, then $b^{\mathcal{PS}} \leq b$ holds; (iii) the third case handles min and sum aggregation over negative numbers; (iv) otherwise, we cannot guarantee any relationship between b and $b^{\mathcal{PS}}$.

Top-K. Recall that the top-k operator returns the k tuples with the smallest values in the **ORDER BY** attributes O. We check whether the condition established for Q_1 imply that the **ORDER BY** attribute values for $D_{\mathcal{PS}}$ are larger than or equal to the ones for D. If that is the case, then tuples that were not part of the top-k answer for D, will not be in top-k on $D_{\mathcal{PS}}$ either. Since no additional attributes are created by this operator, $\Psi_{Q,X,SD}$ is the same as $\Psi_{Q_1,X_1,SD}$.

Duplicate Elimination. Similar with aggregation, instead of checking all group-by attributes are equal on D and $D_{\mathcal{PS}}$, here we check whether the conditions for the input of the duplicate elimination (Q_1) do imply that all the attributes in the schema are equal on D and $D_{\mathcal{PS}}$.

Projection. Since projection does not change the cardinality, only need to require that gc holds for its input. Some additional attributes might be created based on existing attributes by renaming expressions. For these attributes, we could use $\Psi_{Q_1,X_1,SD}$ and $expr({Q_1}^{\mathcal{PS}}) \wedge expr(Q_1)$ to decide the relationship of them between $\Pi_A(Q_1(D_{\mathcal{PS}}))$ and $\Pi_A(Q_1(D))$.

if $X = \emptyset$

$$\Psi_{Q,X,SD} = \bigwedge_{a \in \text{SCH}(Q)} a^{\mathcal{PS}} = a$$
$$gc(Q,X,SD) = \textbf{true}$$

otherwise

$$\Psi_{R,X,SD} = \bigwedge_{a \in \text{SCH}(R)} a^{\mathcal{PS}} = a$$
$$\Psi_{\sigma_\theta(Q_1),X,SD} = \Psi_{\Pi_A(Q_1),X} = \Psi_{Q_1,X_1,SD}$$
$$\Psi_{\delta(Q_1),X,SD} = \Psi_{\tau_{O,C}(Q_1),X,SD} = \Psi_{Q_1,X_1,SD}$$
$$\Psi_{Q_1 \times Q_2,X,SD} = \Psi_{Q_1,X_1,SD} \wedge \Psi_{Q_2,X_2,SD}$$
$$\Psi_{Q_1 \cup Q_2,X,SD} = \bigwedge_{i=1}^{n} (\Psi_{Q_1,X_1,SD} \rightarrow {a_i}^{\mathcal{PS}} = a_i \wedge \Psi_{Q_1,X_2,SD} \rightarrow {b_i}^{\mathcal{PS}} = b_i) \rightarrow {a_i}^{\mathcal{PS}} = a_i$$
$$\textbf{where } \text{SCH}(Q_1) = (a_1, \ldots, a_n) \textbf{ and } \text{SCH}(Q_2) = (b_1, \ldots, b_n)$$

Query Q	$gc(Q,X,SD)$
R	**true**
$\sigma_\theta(Q_1)$	$gc(Q_1,X_1,SD) \wedge (\Psi_{Q_1,X_1,SD} \wedge conds(Q_1) \wedge conds({Q_1}^{\mathcal{PS}}) \wedge \theta^{\mathcal{PS}} \rightarrow \theta)$
$\gamma_{f(a)\rightarrow b;\,G}(Q_1)$	$gc(Q_1,X_1,SD) \wedge (\forall g \in G : \Psi_{Q_1,X_1,SD} \wedge conds(Q_1) \wedge conds({Q_1}^{\mathcal{PS}}) \rightarrow g^{\mathcal{PS}} = g)$
$\delta(Q_1)$	$gc(Q_1,X_1,SD) \wedge (\forall a \in \text{SCH}(Q_1) : \Psi_{Q_1,X_1,SD} \wedge conds(Q_1) \wedge conds({Q_1}^{\mathcal{PS}}) \rightarrow a^{\mathcal{PS}} = a)$
$\Pi_A(Q_1)$	$gc(Q_1,X_1,SD)$
$\tau_{O,C}(Q_1)$	$gc(Q_1,X_1,SD) \wedge (\forall o \in O : \Psi_{Q_1,X_1,SD} \wedge conds(Q_1) \wedge conds({Q_1}^{\mathcal{PS}}) \rightarrow o \leq o^{\mathcal{PS}})$
$Q_1 \cup Q_2$	$gc(Q_1,X_1,SD) \wedge gc(Q_2,X_2,SD)$
$Q_1 \times Q_2$	$gc(Q_1,X_1,SD) \wedge gc(Q_2,X_2,SD)$

(a) $gc(Q, X, SD)$

$$\Psi_{\gamma_{f(a)\rightarrow b;\,G}(Q_1),X,SD} = \begin{cases} \Psi_{Q_1,X_1,SD} \wedge b^{\mathcal{PS}} = b & \textbf{if } \forall x \in X_1 \exists g \in G : conds(Q_1) \rightarrow x = g \\ \Psi_{Q_1,X_1,SD} \wedge b^{\mathcal{PS}} \leq b & \textbf{if } \exists x : x \in X_1 \wedge x \notin G \wedge (f = count \vee (f \in \{sum, max\} \wedge (conds(Q_1) \rightarrow a \geq 0))) \\ \Psi_{Q_1,X_1,SD} \wedge b^{\mathcal{PS}} \geq b & \textbf{if } \exists x : x \in X_1 \wedge x \notin G \wedge (f \in \{sum, min\} \wedge (conds(Q_1) \rightarrow a \leq 0)) \\ \Psi_{Q_1,X_1,SD} & \textbf{otherwise} \end{cases}$$

(b) $\Psi_{Q,X}$

Figure 6: Bottom-up inference of condition $gc(Q,X,SD)$. This condition implies $Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$ if Q is a subquery of the query for which we want to determine sketch safety and $Q(D_{\mathcal{PS}}) = Q(D)$ iff Q is the query for which we want to determine sketch safety.

Cross Product. Only need to require that *gc* holds for both inputs of these operators and the constraints hold for its input also hold for the cross product operator itself.

Union. Only need to require that *gc* holds for both inputs of these operators. and Only constraints that hold in both $\Psi_{Q_1,X_1,SD}$ and $\Psi_{Q_2,X_2,SD}$ hold for the result of the union.

EXAMPLE 9. *Reconsider query Q_{total} with relational algebra: $Q_{total} \coloneqq \tau_{desc,1}(\Pi_{state,sd\cdot-1\rightarrow desc}(\gamma_{state;\,sum(popden)\rightarrow sd}(\text{cities})))$. Since we defined top-k operator order by* **ASC**, *we represents the* **DESC** *by multiplying sd with -1. To determine whether* popden *is a safe attribute for Q_{total} , we calculate $gc(Q_{total}, \{popden\}, SD)$ using the rules from Fig. 6 where SD represents the statics of relation* cities. *For relation* cities, *since popden > 0, then $pred(cities) = popden > 0$, $expr(cities) = \emptyset$, $\Psi_{cities,\{popden\},SD} = popden^{\mathcal{PS}} = popden \wedge city^{\mathcal{PS}} = city \wedge state^{\mathcal{PS}} = state$, and $gc(cities, \{popden\}, SD)$ evaluates to true. Next, $gc(Q_{agg}, \{popden\}, SD)$ evaluates to true, because $\Psi_{cities,\{popden\},SD}$ states that the group-by attribute (state) values are equal: $state^{\mathcal{PS}} = state$. Since sd is computed as a sum over an attribute with positive values, we add $sd \geq sd^{\mathcal{PS}}$ to $\Psi_{Q_{agg},\{popden\},SD}$. The projection multiplies sd with -1. Thus, the constraint $desc = sd \cdot -1$ is added. Finally, for the top-k operator, $desc = sd \cdot -1$ in conjunction with $sd \geq sd^{\mathcal{PS}}$ implies $desc \leq desc^{\mathcal{PS}}$ and $gc(Q_{total}, \{popden\}, SD)$ evaluates to true. Hence, any sketch build on attribute popden is safe for this query.*

8.3 Correctness Proof

We now proceed to prove the correctness of our safety checking algorithm (Thm. 4), i.e., if $gc(Q,X,SD)$ is valid, then X is a set of attributes safe for Q. Before proving our main result we will prove three lemmas that will utilize in this proof. First we prove that, for any query $Q = op(Q_1, \ldots, Q_m)$ where op is an operator, by construction, $\Psi_{Q,X,SD}$ implies $\Psi_{Q_i,X_i,SD}$ and that $gc(Q,X,SD) \Rightarrow gc(Q_i,X_i,SD)$ where X_i is the set of attributes from X that belong to relations accessed by Q_i. This will be used in the proof of following two lemmas. We then prove that $gc(Q,X,SD)$ implies generalized containment: $gc(Q,X,SD) \Rightarrow Q(D_{\mathcal{PS}}) \precsim_{\Psi} Q(D)$. Afterwards, we prove that $gc(Q,X,SD)$ implies that $Q(D)$ is a subset of $Q(D_{\mathcal{PS}})$: $gc(Q,X,SD) \Rightarrow Q(D_{\mathcal{PS}}) \supseteq Q(D)$.

LEMMA 1. *Let D be a database, $Q = op(Q_1, \ldots, Q_n)$ a query, X a set of attributes from the schema of D and the statistics SD of D. Furthermore, let $X_i \subseteq X$ be the subset of X contained in the relations accessed by Q_i. Then, for all $i \in \{1, \ldots, n\}$,*

$$gc(Q,X,SD) \Rightarrow gc(Q_i,X_i,SD) \quad \Psi_{Q,X,SD} \Rightarrow \Psi_{Q_i,X_i,SD}$$

PROOF. For the rules in Fig. 6, $gc(Q,X,SD)$ is based on $gc(Q_1,X_1,SD) \wedge \ldots \wedge gc(Q_n,X_n,SD)$. Thus, this lemma holds. □

Recall in Sec. 8.1 we defined general containment to model the relationship between (intermediate) results of $Q(D_{\mathcal{PS}})$ and $Q(D)$ and designed $gc(Q,X,SD)$ rules in Fig. 6 to trace the evolution of the general containment. Thus now we prove that $gc(Q,X,SD) \Rightarrow Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$.

LEMMA 2. *Let Q be a query, D be a database, SD the statics of D, and $X = \bigcup_1^n X_i$ a set of attributes where each X_i belongs to a relation R_i accessed by Q such that $R_i \neq R_j$ for $i \neq j$. Given a set of provenance sketches $\mathcal{PS} = \{\mathcal{P}_i\}$ for Q over D with respect to a set of range partitions $\{F_{\mathcal{R}_i,X_i}(R_i)\}$, then*

$$gc(Q,X,SD) \Rightarrow Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$$

PROOF. We prove this by induction. For convenience, given tuple $t \in Q(D_{\mathcal{PS}})$ and $t' \in Q(D)$, we say t *mapped* with t', if $(t,t') \in \mathcal{M}$ where $\mathcal{M} \subseteq Q(D_{\mathcal{PS}}) \times Q(D)$. Also, given a set of attributes $A = \{a_1, \ldots, a_n\}$ and $A' = \{a'_1, \ldots, a'_n\}$, for convenience, we say $A = A'$ if $a_1 = a'_1, \ldots, a_n = a'_n$.

Base case: We start from the relation access R. Assume $R_{\mathcal{PS}}$ represents the provenance sketches of R with respect to a set of range partitions $\{F_{\mathcal{R},X}(R)\}$, then $R_{\mathcal{PS}} \subseteq R$, which is a special case of $R_{\mathcal{PS}} \precsim_{\Psi_{R,X,SD}} R$ where Ψ only contains equalties for each column between the schema of $R_{\mathcal{PS}}$ and R. Therefore, $R_{\mathcal{PS}} \precsim_{\Psi_{R,X,SD}} R$.

Inductive step: Assume query Q_n, a sub query of Q, with depth less than or equal to n. Here we use the depth to represent the levels in the relational algebra tree of a query, e.g., Q_1 represents the relation access R. Assume we have proven that $gc(Q_n,X_n,SD) \Rightarrow Q_n(D_{\mathcal{P}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$. Based on the induction hypothesis, we have to prove $gc(Q_{n+1},X_{n+1},SD) \Rightarrow Q_{n+1}(D_{\mathcal{P}}) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}}$

$Q_{n+1}(D)$. For notational convenience, we use $op_n(op_{n+1})$ represent any operator of depth n $(n{+}1)$, i.e., $Q_{n+1} = op_{n+1}(Q_n)$. From Lem. 1, $gc(Q_{n+1}, X_{n+1}, SD) \Rightarrow gc(Q_n, X_n, SD)$. Then based on the assumption, $Q_n(D_{\mathcal{P}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$. For join operator, we assume the left and right input of op_{n+1} are $Q_{nL}(D)$ and $Q_{nR}(D)$ respectively, then $gc(Q_{nL}, X_{nL}, SD) = \textbf{true}$ and $gc(Q_{nR}, X_{nR}, SD) = \textbf{true}$, and $Q_{nL}(D_{\mathcal{P}}) \precsim_{\Psi_{Q_{nL},X_{nL},SD}} Q_{nL}(D)$ and $Q_{nL}(D_{\mathcal{P}}) \precsim_{\Psi_{Q_{nL},X_{nL},SD}} Q_{nL}(D)$. Now we consider different op_{n+1}:

$\underline{\sigma_\theta}$: Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, then assume tuple $t \in Q_n(D_{\mathcal{PS}})$, then there have to be a tuple $t' \in Q_n(D)$ such that t mapped with t' and $(t, t') \models \Psi_{Q_n,X,SD}$. Then t satisfies $conds({Q_n}^{\mathcal{PS}})$ and t' satisfies $conds(Q_n)$. Assume after σ_θ, t exists in the result of $\sigma_\theta(Q_n(D_{\mathcal{PS}}))$, that is $t \in \sigma_\theta(Q_n(D_{\mathcal{PS}}))$, thus t satisfies θ. Since $\Psi_{Q_n,X,SD} \wedge conds(Q_n) \wedge conds({Q_n}^{\mathcal{PS}}) \wedge \theta^{\mathcal{PS}} \to \theta$ in Fig. 6, then t' satisfies θ, i.e., $t' \in \sigma_\theta(Q_n(D))$. Therefore, $op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,X,SD}$ and $X_{n+1} = X_n$.

$\underline{\delta}$: Assume $\exists t \in \delta(Q_n(D_{\mathcal{PS}}))$, then $t \in Q_n(D_{\mathcal{PS}})$. Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, then there have to be a tuple $t' \in Q_n(D)$ such that t mapped with t' and $(t, t') \models \Psi_{Q_n,X,SD}$. Thus, in Fig. 6, the duplicate removal rule guarantees that $\forall a \in \mathrm{SCH}(Q_n)$: $\Psi_{Q_n,X,SD} \wedge conds(Q_n) \wedge conds({Q_n}^{\mathcal{PS}}) \Rightarrow a^{\mathcal{PS}} = a$, that is $t = t'$. And because $t' \in \delta(Q_n(D))$, $op_{n+1}(Q_n(D_{\mathcal{PS}})) \subseteq op_{n+1}(Q_n(D))$. Then same with the base case which is a special case of the general containment, thus $op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where $\Psi_{Q_{n+1},X_{n+1},SD}$ only contains equalities.

$\underline{\Pi_A}$: Similarly, assume $\exists t_{proj} \in \Pi_A(Q_n(D_{\mathcal{PS}}))$. Then, there have to be a tuple $t \in Q_n(D_{\mathcal{PS}})$ such that $\Pi_A(\{t\}) = \{t_{proj}\}$. Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, then there have to be a tuple $t' \in Q_n(D)$ such that t mapped with t' and $(t, t') \models \Psi_{Q_n,X,SD}$. That it, there have to be a tuple tt' satisfies $\{t'_{proj}\} = \Pi_A(\{t'\})$ such that (t_{proj}, t'_{proj}) satisfies $\Psi_{Q_n,X_n,SD}$. Thus, $op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,X_n,SD}$.

$\underline{Q_{Ln}(D) \cup Q_{Rn}(D)}$: Assume $\exists t_L \in Q_{Ln}(D_{\mathcal{PS}})$, since $Q_{Ln}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{Ln},X_{Ln,SD}}} Q_{Ln}(D)$, then there have to be a tuple $t'_L \in Q_{Ln}(D)$ such that t_L mapped with t'_L and $(t_L, t'_L) \models \Psi_{Q_{Ln},X_{Ln,SD}}$. After union, $t_L \in Q_{Ln}(D_{\mathcal{PS}}) \cup Q_{Rn}(D_{\mathcal{PS}})$ and $t'_L \in Q_{Ln}(D) \cup Q_{Rn}(D)$. Assume $\exists t_R \in Q_{rn}(D_{\mathcal{PS}})$, similarly, we could get that there have to be a tuple $t'_R \in Q_{Rn}(D)$ such that t_R mapped with t'_R and $(t_R, t'_R) \models \Psi_{Q_{Rn},X_{Rn},SD}$. After union, $t_R \in Q_{Ln}(D_{\mathcal{PS}}) \cup Q_{Rn}(D_{\mathcal{PS}})$ and $t'_R \in Q_{Ln}(D) \cup Q_{Rn}(D)$. Then $\{t_L, t_R\} \subseteq Q_{Ln}(D_{\mathcal{PS}}) \cup Q_{Rn}(D_{\mathcal{PS}})$ and $\{t'_L, t'_R\} \subseteq Q_{Ln}(D) \cup Q_{Rn}(D)$. So now the generalized containment will only hold for the common part between $\Psi_{Q_{Ln},X_{Ln,SD}}$ and $\Psi_{Q_{Rn},X_{Rn},SD}$. That is,
$op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where
$\Psi_{Q_{n+1},X_{n+1},SD} = \bigwedge_{a=a' \in \Psi_{Q_{Ln},X_{Ln,SD}} \wedge a=a' \in \Psi_{Q_{Rn},X_{Rn},SD}} a = a'$.

$\underline{\gamma_{f(a)\to b;G}}$: Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, then assume $\exists t \in Q_n(D_{\mathcal{PS}})$, there has to be a tuple $t' \in Q_n(D)$ such that t mapped with t' and $(t, t') \models \Psi_{Q_n,X,SD}$. And because of the aggregation rule in Fig. 6a which keeps that $\forall g \in G : g^{\mathcal{PS}} = g$, then after aggregation, we will still get $Q_{n+1}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} Q_{n+1}(D)$ on $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,X_n,SD}$. To learn the relation between $b^{\mathcal{PS}}$ and b, we consider different cases: **CASE 1:** $\forall x \in X_1 \exists g \in G$: $conds(Q_1) \to x = g$. (i) We first prove that $b^{\mathcal{PS}} = b$ when all the operators under op_{n+1} are monotone operators. Based on rules in Fig. 6, only aggregation could generate inequality in $\Psi_{Q_i,X_i,SD}$ $(1 \le i \le n)$, then $\Psi_{Q_n,X_n,SD}$ only contains equalities, so $Q_n(D_{\mathcal{PS}}) \subseteq Q_n(D)$. Now we discuss $Q_{n+1}(D_{\mathcal{PS}})$ and $Q_{n+1}(D)$. To make $Q_{n+1}(D_{\mathcal{PS}}) \subseteq Q_{n+1}(D)$, there should not exist the case that $\exists t_1 \in D_{\mathcal{PS}}$ and $\exists t_2 \in D - D_{\mathcal{PS}}$ where $t_1.x = t_2.x$. However, this case will not happen since $\forall x \in X_{n+1} \exists g \in G : conds(Q_{n+1}) \to x = g$. Thus, $Q_{n+1}(D_{\mathcal{PS}}) \subseteq Q_{n+1}(D)$ and $b^{\mathcal{PS}} = b$. (ii) We next prove that $b^{\mathcal{PS}} = b$ when there is only one aggregation under op_{n+1}. We represents this aggregation as $\gamma_{f_1(a_1)\to b_1;G_1}$ and use Q_{G_1} represent the subquery rooted at this aggregation. By continuing apply Lem. 1, we could get $Q_{G_1}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{G_1},X_{G_1},SD}} Q_{G_1}(D)$ since only monotone operators under this aggregation. For convenience we assume that all columns' name appeared in a query are unique. Because $X_{n+1} \subseteq \mathrm{SCH}(D)$ and $\forall x \in X_{n+1} \exists g \in G : conds(Q_n) \to x = g$, then $\forall x \in X_{G_1} \exists g \in G_1 : x = g$, otherwise, we will lose some columns used in X_{n+1} before reaching op_{n+1}. Thus, after $\gamma_{f_1(a_1)\to b_1;G_1}$, we get ${b_1}^{\mathcal{PS}} = b_1$. Then, $\Psi_{Q_{G_1},X_{G_1},SD}$ would only contain equalities and thus $Q_{G_1}(D_{\mathcal{PS}}) \subseteq Q_{G_1}(D)$. And only monotone operators exist between ${}_{F_1(a)\to f_1}\gamma_{G_1}$ and op_{n+1}, which would not change the Ψ, thus $\Psi_{Q_n,X_n,SD}$ would only contain equalities. Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, $Q_n(D_{\mathcal{PS}}) \subseteq Q_n(D)$. Then, similar to (i), we get $b^{\mathcal{PS}} = b$. (iii) At last, we prove that $b^{\mathcal{PS}} = b$ when there are many aggregations under op_{n+1}. Assume these aggregations are $\gamma_{f_1(a_1)\to b_1;G_1}, \ldots, \gamma_{f_k(a_k)\to b_k;G_k}$ from bottom to up respectively. The matched subqueries are $Q_{G_1}, \ldots, Q_{G_k}$. Since $X_{n+1} \subseteq \mathrm{SCH}(D)$ and $\forall x \in X_{n+1} \exists g \in G : conds(Q_n) \to x = g$, then $\forall x \in X_{G_1} \exists g \in G_1 :\to x = g, \ldots, \forall x \in X_{G_k} \exists g \in G_k :\to x = g$, otherwise, we will lose some columns in X before reaching op_{n+1}. Then by reapplying (ii), we would get $b^{\mathcal{PS}} = b$. Note that for non-monotone operators, we only consider aggregation in this paper. Thus, $X_{n+1} = X_n \wedge b^{\mathcal{PS}} = b$. **CASE 2:** $\exists x : x \in X_1 \wedge x \notin G \wedge (f = count \vee (f \in \{sum, max\} \wedge (conds(Q_1) \to a \ge 0)))$. Recall we have proven that $Q_{n+1}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} Q_{n+1}(D)$ where $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,X_n,SD}$. Assume exists a pair of matched tuples (t, t') in $Q_{n+1}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} Q_{n+1}(D)$ where $\{t\} = op_{n+1}(Q_n(\{t_1, \ldots, t_m\}))$ and $\{t'\} = op_{n+1}(Q_n(\{t'_1, \ldots, t'_n\}))$. That is, $\{t_1, \ldots, t_m\} \subseteq Q_{n+1}(D_{\mathcal{PS}})$ and $\{t'_1, \ldots, t'_n\} \subseteq Q_{n+1}(D)$. If $\exists x : x \in X_n \wedge x \notin G$, then $m \le n$. Since $f = count \vee (f \in \{sum, max\} \wedge (conds(Q_n) \to a \ge 0))$, $b^{\mathcal{PS}} \le b$. Thus, $X_{n+1} = X_n \wedge b^{\mathcal{PS}} \le b$. **CASE 3:** $\exists x : x \in X_1 \wedge x \notin G \wedge (f \in \{sum, min\} \wedge (conds(Q_1) \to a \le 0))$. Similar to CASE 2, if $\exists x : x \in X_n \wedge x \notin G$, then $m \le n$. Since $f \in \{sum, min\} \wedge (conds(Q_n) \to a \le 0)$, $b^{\mathcal{PS}} \ge b$. Thus, $X_{n+1} = X_n \wedge b^{\mathcal{PS}} \ge b$. **CASE 4:** Otherwise, we are unable to decide the relationship between $b^{\mathcal{PS}}$ and b. Thus, $op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,x}$ and $X_{n+1} = X_n$.

$\underline{\tau_{O,C}}$: Assume $\exists t \in Q_n(D_{\mathcal{PS}})$, since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, then there have to be a tuple $t' \in Q_n(D)$ such that t mapped with t' and $(t, t') \models \Psi_{Q_n,X,SD}$. Assume t' satisfies the `LIMIT` condition, i.e., $t' \in \tau_{O,C}(Q_n(D))$. Since $\forall o \in O : \Psi_{Q_1,X_1,SD} \wedge conds({Q_1}^{\mathcal{PS}}) \wedge conds(Q_1) \to o \le o^{\mathcal{PS}}$, then t has to satisfy the `LIMIT` condition and thus $t \in \tau_{O,C}(Q_n(D_{\mathcal{PS}}))$. Thus, $op_{n+1}(Q_n(D_{\mathcal{PS}})) \precsim_{\Psi_{Q_{n+1},X_{n+1},SD}} op_{n+1}(Q_n(D))$ where $\Psi_{Q_{n+1},X_{n+1},SD} = \Psi_{Q_n,X_n,SD}$.

$\underline{Q_{Ln}(D) \times Q_{Rn}(D)}$: Let $Q_{Ln}(D)$ and $Q_{Rn}(D)$ be the left and right children of op_{n+1} respectively. Since $gc(Q_{n+1}, X_{n+1}, SD) = \textbf{true}$, based on Lem. 1, $gc(Q_{Ln}, X_{Ln}, SD) = \textbf{true}$ and $gc(Q_{Rn}, X_{Rn}, SD) =$

true. Thus $Q_{Ln}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{Ln},X_{Ln},SD}} Q_{Ln}(D)$ and $Q_{Rn}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{Ln},X_{Rn},SD}} Q_{Rn}(D)$. Then $Q_{Ln}(D_{\mathcal{PS}}) \times Q_{Rn}(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_{Ln}\times Q_{Rn},X_{Q_{Ln}\times Q_{Rn}},SD}} Q_{Ln}(D) \times Q_{Rn}(D)$ where $\Psi_{Q_{Ln}\times Q_{Rn},X_{Q_{Ln}\times Q_{Rn}},SD} = \Psi_{Q_{Ln},X_{Ln},SD} \wedge \Psi_{Q_{Rn},X_{Ln},SD}$ and $X_{Q_{Ln}\times Q_{Rn}} = X_{Q_{Ln}} \wedge X_{Q_{Rn}}$.

□

In Lem. 2 we proved that $gc(Q,X,SD) \Rightarrow Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$. Since $D_{\mathcal{PS}} \supseteq P(Q,D)$, now we reason about that non-provenance tuples in $D_{\mathcal{PS}}$ would not result in any result tuple which is different with the result in $Q(D)$. That is, if we generate a tuple in $Q(D)$, we would also generate this tuple in $Q(D_{\mathcal{PS}})$.

LEMMA 3. *Let Q be a query, D be a database, SD be the statics of D and $X = \bigcup_1^n X_i$ a set of attributes where each X_i belongs to a relation R_i accessed by Q such that $R_i \neq R_j$ for $i \neq j$. Given a set of provenance sketches $\mathcal{PS} = \{\mathcal{P}_i\}$ for Q over D with respect to a set of range partitions $\{F_{\mathcal{R}_i,X_i}(R_i)\}$, then*

$$gc(Q,X,SD) \Rightarrow Q(D) \subseteq Q(D_{\mathcal{PS}})$$

PROOF. $Q(D) \subseteq Q(D_{\mathcal{PS}})$ represents that if exists a tuple $t \in Q(D)$, then $t \in Q(D_{\mathcal{PS}})$. That is, we need to prove that $gc(Q,X,SD) \wedge \exists t \in Q(D) \Rightarrow t \in Q(D_{\mathcal{PS}})$. In the following, we prove this by induction through proving this formula holds for each subquery of Q. However, not all the subquery's result satisfy this formula, since there is a implicit condition $P(t) \subseteq P(Q,D)$ under $gc(Q,X,SD) \wedge \exists t \in Q(D) \Rightarrow t \in Q(D_{\mathcal{PS}})$. Recall that we use $P(Q,D)$ to represent the provenance of Q over D, for convenience, we use $P(t)$ to represent the provenance to derive tuple t. Then the meaning is that only the intermediate result tuples which contribute to the final result $Q(D)$ would satisfy this formula. Thus, what we need to prove is that $gc(Q,X,SD) \wedge \exists t \in Q(D) : P(t) \subseteq P(Q,D) \Rightarrow t \in Q(D_{\mathcal{PS}})$.

Base case: When Q is table access operator R, if $t \in R$, $P(t) = \{t\}$. Recall that provenance sketches are a superset of provenance, that is $P(t) \subseteq R_{\mathcal{PS}}$. Thus $t \in R_{\mathcal{PS}}$.

Inductive step: Assume query Q_n, a sub query of Q, with depth less than or equal to n. Here we use *depth* to represent the levels in the relational algebra format of a query, e.g., Q_1 represents the base table R with depth 1. Assume we have proven that $gc(Q_n,X,SD) \wedge \exists t \in Q_n(D) : P(t) \subseteq P(Q,D) \Rightarrow t \in Q_n(D_{\mathcal{PS}})$. Based on the induction hypothesis, we have to prove that the same holds for sub query Q_{n+1} with depth less than or equal to $n+1$, that is, we need to prove $gc(Q_{n+1},X_{n+1},SD) \wedge \exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D) \Rightarrow t \in Q_{n+1}(D_{\mathcal{PS}})$. For notational convenience, we use $op_n(op_{n+1})$ represent any operator of depth n $(n+1)$, i.e., $Q_{n+1} = op_{n+1}(Q_n)$.

Since $gc(Q_{n+1},X_{n+1},SD) =$ **true**, base on Lem. 1, then $gc(Q_n,X_n,SD) =$ **true**. For join operator, we assume the left and right input of op_{n+1} are $Q_{nL}(D)$ and $Q_{nR}(D)$ respectively, then $gc(Q_{nL},X_{nL},SD) =$ **true** and $gc(Q_{nR},X_{nR},SD) =$ **true**.

Now we discuss different op_{n+1}:

For Π_A: Assume $\exists t_{proj} \in Q_{n+1}(D) : P(t_{proj}) \subseteq P(Q,D)$, then there have to be a tuple $t \in Q_n(D)$ such that $\Pi_A(\{t\}) = \{t_{proj}\}$. Since $P(t_{proj}) = P(t)$, $P(t) \subseteq P(Q,D)$. And because $gc(Q_n,X_n,SD) =$ **true**, based on assumption, $t \in Q_n(D_{\mathcal{PS}})$. Thus, from $Q_{n+1}(D_{\mathcal{PS}}) = \Pi_A(Q_n(D_{\mathcal{PS}}))$, $t \in Q_n(D_{\mathcal{PS}})$ and $\Pi_A(\{t\}) = \{t_{proj}\}$, we get $t_{proj} \in Q_{n+1}(D_{\mathcal{PS}})$.

For σ_θ: Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$, since $\sigma_\theta(\{t\}) = \{t\}$, $t \in Q_n(D)$. Because $gc(Q_n,X_n,SD) =$ **true**, based on assumption, $t \in Q_n(D_{\mathcal{PS}})$. Thus, $t \in Q_{n+1}(D_{\mathcal{PS}})$.

For δ: Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$, there have to exist at least one tuple $t' \in Q_n(D)$ such that $t = t'$, that is $P(t') \subseteq P(t)$, thus $P(t') \subseteq P(Q,D)$. And because $gc(Q_n,X_n,SD) =$ **true**, based on assumption, $t' \in Q_n(D_{\mathcal{PS}})$, that it $t \in Q_n(D_{\mathcal{PS}})$. Thus, $t \in Q_{n+1}(D_{\mathcal{PS}})$.

$\tau_{O,C}$: Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$, then $t \in Q_n(D)$. And because $gc(Q_n,X_n,SD) =$ **true**, based on assumption, $t \in Q_n(D_{\mathcal{PS}})$. Let T represent all these t, that is $T \subseteq Q_{n+1}(D)$. Now we decide whether $T \subseteq Q_{n+1}(D_{\mathcal{PS}})$. Recall Lem. 2 proved that $gc(Q_n,X_n,SD) \Rightarrow Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$. Also we have $T \subseteq Q_n(D_{\mathcal{PS}})$, $T \subseteq Q_n(D)$. Since $\forall o \in O : o \leq o^{\mathcal{PS}}$ from the top-k rule in Fig. 6, $\forall t' \in Q_n(D_{\mathcal{PS}}) - T$ and $\forall t'' \in T$, $t'.O \geq t''.O$. Thus, if $T \subseteq Q_{n+1}(D)$, then $T \subseteq Q_{n+1}(D_{\mathcal{PS}})$.

$Q_{Ln}(D) \cup Q_{Rn}(D)$: Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$, then $t \in Q_{Ln}(D)$ or $t \in Q_{Rn}(D)$. And because $gc(Q_{nL},X_{nL},SD) =$ **true** and $gc(Q_{nR},X_{nR},SD) =$ **true**, based on assumption, $t \in Q_{Ln}(D_{\mathcal{PS}})$ or $t \in Q_{Rn}(D_{\mathcal{PS}})$. Thus, $t \in Q_{n+1}(D_{\mathcal{PS}})$.

For $\times$:Let $Q_{nL}(D)$ and $Q_{nR}(D)$ represent the left and right input of op_{n+1} respectively. Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$, then there have to be a tuple $t'_L \in Q_{nL}(D)$ and a tuple $t'_R \in Q_{nR}(D)$ such that $\{t'_L\} op_{n+1} \{t'_R\} = \{t\}$. And $P(t'_L) \cup P(t'_R) = P(t)$, thus $P(t'_L) \subseteq P(Q,D)$ and $P(t'_R) \subseteq P(Q,D)$. And because $gc(Q_{nL},X_{nL},SD) =$ **true** and $gc(Q_{nR},X_{nR},SD) =$ **true**, based on assumption, $t'_L \in Q_{nL}(D_{\mathcal{PS}})$ and $t'_R \in Q_{nR}(D_{\mathcal{PS}})$. Thus $t \in Q_{n+1}(D_{\mathcal{PS}})$.

For $\gamma_{f(a)\rightarrow b;G}$: Assume $\exists t \in Q_{n+1}(D) : P(t) \subseteq P(Q,D)$ and $\{t_1,...,t_m\} \subseteq Q_n(D)$ such that $op_{n+1}(\{t_1,...,t_m\}) = \{t\}$. Then $P(t_1) \cup \ldots \cup P(t_m) = P(t)$, thus $P(t_1) \subseteq P(Q,D), \ldots, P(t_m) \subseteq P(Q,D)$. And because $gc(Q_n,X_n,SD) =$ **true**, based on assumption, $\{t_1,...,t_m\} \subseteq Q_n(D_{\mathcal{PS}})$. To let $t \in Q_{n+1}(D_{\mathcal{PS}})$, we have to keep that not exists a set of tuples $T \subseteq Q_n(D_{\mathcal{PS}})$ such that $op_{n+1}(\{t_1,...,t_m\} \cup T) = \{t'\}$ and $t' != t$ where $t' \in Q_{n+1}(D)$. Then, the question transforms to prove no such T in $Q_n(D_{\mathcal{PS}}) - \{t_1,...,t_m\}$. Since $gc(Q_n,X_n,SD) =$ **true**, based on Lem. 2, then $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$. Assume op_{n+1} groups on columns $\{a_1,\ldots,a_k\}$ and for $\{t_1,...,t_m\}$, the values on these columns are $\{v_1,\ldots,v_k\}$. Then, that is in $Q_n(D) - \{t_1,...,t_m\}$, there are no tuples on these columns have same values with $\{v_1,\ldots,v_k\}$. Because $gc(Q_{n+1},X_{n+1},SD) =$ **true**, the aggregation rule in Fig. 6a are satisfied, then we know $\forall g \in G : g^{\mathcal{PS}} = g$, that is ${a_1}^{\mathcal{PS}} = a_1, \ldots, {a_k}^{\mathcal{PS}} = a_k$. Since $Q_n(D_{\mathcal{PS}}) \precsim_{\Psi_{Q_n,X,SD}} Q_n(D)$, if no tuples in $Q_n(D) - \{t_1,...,t_m\}$ have same values with $\{v_1,\ldots,v_k\}$ on $\{a_1,\ldots,a_k\}$, then there no tuples in $Q_n(D_{\mathcal{PS}}) - \{t_1,...,t_m\}$ have the same values with $\{v_1,\ldots,v_k\}$ on $\{a_1,\ldots,a_k\}$. Thus, T does not exist, we have $t \in Q_{n+1}(D_{\mathcal{PS}})$. □

In Lem. 2 we proved that $Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$ and Lem. 3 keeps the matched tuples are equal, that is, $Q(D_{\mathcal{PS}}) \subseteq Q(D)$. Then, we have $Q(D_{\mathcal{PS}}) = Q(D)$.

THEOREM 4 ($gc(Q,X,SD)$ IMPLIES SAFETY OF X). *Let Q be a query, D be a database, SD be the statics of SD and $X = \bigcup_1^n X_i$ a set of attributes where each X_i belongs to a relation R_i accessed by Q such that $R_i \neq R_j$ for $i \neq j$. If $gc(Q,X,SD)$ holds, then X is a safe set of attributes for Q.*

Proof. Recall that a set of attribute X is safe, if for databases D and all sets of provenance sketches $\mathcal{PS} = \{\mathcal{P}_i\}$ for Q over D with respect to a set of range partitions $\{F_{\mathcal{R}_i,X_i}(R_i)\}$, we have

$$gc(Q, X, SD) \Rightarrow Q(D_{\mathcal{PS}}) = Q(D)$$

Since $gc(Q, X, SD) = \textbf{true}$, based on Lem. 2, $Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$; based on Lem. 3, $Q(D_{\mathcal{PS}}) \supseteq Q(D)$. Thus, for each pair of matched tuples (t, t') in $Q(D_{\mathcal{PS}}) \precsim_{\Psi_{Q,X,SD}} Q(D)$ where $t \in Q(D_{\mathcal{PS}})$ and $t' \in Q(D)$, $t = t'$, that is, $Q(D_{\mathcal{PS}}) \subseteq Q(D)$. Thus, $Q(D_{\mathcal{PS}}) = Q(D)$. □

Furthermore, since our rules are independent of the number of fragments, adding fragments to a safe sketch, the resulting sketch is guaranteed to be safe too.

9 REUSING PROVENANCE SKETCHES FOR PARAMETERIZED QUERIES

Given a set of accurate provenance sketches $\mathcal{PS}$ captured for a query Q, we would like to be able to use $\mathcal{PS}$ to answer future queries Q'. To determine whether this is possible, we need to determine whether $D_{\mathcal{PS}}$ is sufficient for Q'. This is similar to checking query containment which is known to be undecidable for the class of queries we are interested in [19, 58, 83]. Here we focus on developing a solution for restricted version of this problem: reusing sketches across multiple instances of a parameterized query [10]. Given the prevalence of parameterized queries in applications and reporting tools that access a database, this is an important special case. Note that even for ad hoc analytics, it is common that query patterns repeat if the number of queries is sufficiently large. While such queries are typically not expressed as parameterized queries we can treat them as such by replacing all constants in selection conditions with parameters. Note that the problem studied in this section can also be interpreted as a generalization of safety checking for sketches with the difference that we determine the safety of a sketch for a different query instead of for the query it was captured for. The major result of this section is a sufficient condition for checking whether a sketch can be reused that is rooted in the safety conditions from Sec. 8.

Let $\mathbb{P}$ be a countable set of variables called parameters. A *parameterized query* $\mathcal{T}[\vec{p}]$ for $\vec{p} = (p_1, \ldots, p_n)$ and $p_i \in \mathbb{P}$ is a relational algebra expression where conditions of selections may refer to parameters from the set $\{p_i\}$. We assume that each parameter from $\vec{p}$ is referenced at least once by $\mathcal{T}[\vec{p}]$. A parameter binding $\vec{v}$ for $\mathcal{T}[\vec{p}]$ is a vector of constants, one for each parameter p_i from $\vec{p}$. The *instance* $\mathcal{T}[\vec{v}]$ of $\mathcal{T}[\vec{p}]$ for $\vec{v}$ is the query resulting from substituting p_i with v_i in $\mathcal{T}$. For instance, the parameterized SQL query `SELECT * FROM R WHERE a < $1` can be written as $\mathcal{T}[p_1] = \sigma_{a<p_1}(R)$. We define the **sketch reusability problem** as: given a parameterized query $\mathcal{T}$, two instances Q and Q' for $\mathcal{T}$, and a safe set of provenance sketches $\mathcal{PS}$ for Q, determine whether $D_{\mathcal{PS}}$ is sufficient for Q'. In the remainder of this section we develop a *sufficient* condition for sketch reusability. Before presenting our sufficient condition, we first state three lemmas that we will use to develop this condition. First, observe that the same sets of attributes are safe for all instances of a parameterized query.

Lemma 4. *Let Q and Q' be two instances of a parameterized query $\mathcal{T}$, then any set of attributes X that is safe for Q is safe for Q'.*

Proof. The definition of safety from Sec. 8 does not consider constants in selection conditions. Since Q and Q' only differ in such constants, any sketch type that is safe for Q is also safe for Q'. □

Furthermore, adding additional fragments to a safe sketch $\mathcal{P}$ for a query Q yields a safe sketch (Sec. 8).

Lemma 5. *Let $\mathcal{P}$ be a safe sketch for a query Q. Then any sketch $\mathcal{P}' \supseteq \mathcal{P}$ is safe for Q.*

Proof. Since our safety rules introduced in sec. 8 only care about which columns we do range partition on, that is if we say an accurate provenance sketch safe, any superset of it is still safe. □

Recall that accurate provenance sketches are sketches which do only contain ranges whose fragments contain provenance. Consider a database D and two queries Q and Q' and denote the provenance of Q (Q') as $P(Q, D)$ ($P(Q', D)$). Furthermore, consider two sets of accurate provenance sketches $\mathcal{PS}$ and $\mathcal{PS}'$ build over the same attributes X and partitions where $\mathcal{PS}$ ($\mathcal{PS}'$) is a sketch for Q (Q'). If $P(Q, D) \supseteq P(Q', D)$ then $\mathcal{PS} \supseteq \mathcal{PS}'$ and, thus, also $D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}'}$.

Lemma 6. *Consider two queries Q and Q' and database D with n relation and let $\mathcal{PS} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ and $\mathcal{PS}' = \{\mathcal{P}'_1, \ldots, \mathcal{P}'_m\}$ be two set of accurate provenance sketches for Q (Q') that are both based on the same set of range partitions $\{F_{\mathcal{R}_1,a_1}(R_1), \ldots, F_{\mathcal{R}_m,a_m}(R_m)\}$ where $\{R_1, \ldots, R_m\} \subseteq \mathrm{SCH}(D)$. We have,*

$$P(Q, D) \supseteq P(Q', D) \Rightarrow \mathcal{PS} \supseteq \mathcal{PS}' \wedge D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}'}$$

Proof. Since $P(Q, D) \supseteq P(Q', D)$, for each tuple $t \in P(Q', D)$, $t \in P(Q, D)$. Assume $t \in R_r$ where $R \in D$, then for each $r \in \mathcal{P}'$, $r \in \mathcal{P}$, that is $\mathcal{P} \supseteq \mathcal{P}'$. Since it is hold for each $\mathcal{P}$, we say $\mathcal{PS} \supseteq \mathcal{PS}'$ and thus $D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}'}$. □

These lemmas imply that a provenance sketch for any instance Q of a parameterized query $\mathcal{T}$ is safe for another instance Q' of $\mathcal{T}$, if $P(Q, D) \supseteq P(Q', D)$. We refer to this as *provenance containment*. In the following we develop a sufficient condition that guarantees provenance containment for all input databases D. We again use an SMT solver similar to how we checked safety in Sec. 8. Our condition consists of two parts: $uconds(Q', Q)$ (shown below) and $ge(Q', Q)$. Condition ge (Fig. 7) serves a similar purpose as gc in our safety condition. It is defined recursively over the structure of a query and we construct a formula $\Psi_{Q',Q}$ over comparisons between attributes from Q and Q' such that ge (together with the condition $uconds$ explained below) implies general containment ($Q'(D) \precsim_{\Psi_{Q',Q}} Q(D)$). Furthermore, we demonstrate that ge in conjunction with $uconds$ implies provenance containment and, thus, safety of $\mathcal{PS}$ for Q'. We will use a to refer to attributes from Q and a' to refer to the corresponding attribute from Q'. Similarly, if θ is a condition in Q, then θ' denotes the corresponding condition in Q'.

The main difference of ge and gc is that we are now dealing with two different queries instead of one query. Furthermore, we do not access D in ge. The selection conditions of the two queries that restrict values of an attribute may be spread over multiple operators in these queries. It is possible that the conditions of all selections of Q' imply the conditions of all selections of Q even though this does not hold for all individual selections of these two queries. As a trivial example consider $Q = \sigma_{a=20}(\sigma_{a>30}(R))$ and $Q' = \sigma_{a=20}(\sigma_{a>10}(R))$.

Query $\mathcal{T}$	$ge(Q', Q)$
R	**true**
$\sigma_\theta(\mathcal{T}_1)$	$ge(Q'_1, Q_1)$
$\gamma_{f(a)\rightarrow b;\,G}(\mathcal{T}_1)$	$ge(Q'_1, Q_1) \wedge (\forall g \in G : \Psi_{Q'_1,Q_1} \wedge conds(Q_1) \wedge conds(Q'_1) \rightarrow g = g')$
$\delta(\mathcal{T}_1)$	$ge(Q'_1, Q_1) \wedge (\forall a \in \mathrm{SCH}(Q_1) : \Psi_{Q'_1,Q_1} \wedge conds(Q_1) \wedge conds(Q'_1) \rightarrow a = a')$
$\Pi_A(\mathcal{T}_1)$	$ge(Q'_1, Q_1)$
$\mathcal{T}_1 \cup \mathcal{T}_2$	$ge(Q'_1, Q_1) \wedge ge(Q'_2, Q_2)$
$\mathcal{T}_1 \times \mathcal{T}_2$	$ge(Q'_1, Q_1) \wedge ge(Q'_2, Q_2)$

(a) $ge(Q', Q)$

$$\Psi_{R',R} = \bigwedge\nolimits_{a \in \mathrm{SCH}(R)} a = a'$$

$$\Psi_{\sigma_{\theta'}(Q'_1),\sigma_\theta(Q_1)} = \Psi_{Q'_1,Q_1}$$

$$\Psi_{\Pi_A(Q'_1),\Pi_A(Q_1)} = \Psi_{Q'_1,Q_1}$$

$$\Psi_{\delta(Q'_1),\delta(Q_1)} = \Psi_{Q'_1,Q_1}$$

$$\Psi_{Q'_1 \times' Q'_2, Q_1 \times Q_2} = \Psi_{Q'_1,Q_1} \wedge \Psi_{Q'_2,Q_2}$$

$$\Psi_{Q'_1 \cup' Q'_2, Q_1 \cup Q_2} = \bigwedge\nolimits_{i=1}^{n} (\Psi_{Q'_1,Q_1} \rightarrow a_i = a'_i \wedge \Psi_{Q'_2,Q_2} \rightarrow b_i = b'_i) \rightarrow a_i = a'_i \textbf{ where } \mathrm{SCH}(Q_1) = (a_1, \ldots, a_n) \textbf{ and } \mathrm{SCH}(Q_2) = (b_1, \ldots, b_n)$$

$$\Psi_{\gamma'_{f(a)\rightarrow b;\,G}(Q'_1),\gamma_{f(a)\rightarrow b;\,G}(Q_1)} = \begin{cases} \Psi_{Q'_1,Q_1} \wedge b = b' & \text{if } \textcircled{1} \wedge \textcircled{2} \\ \Psi_{Q'_1,Q_1} \wedge b \leq b' & \text{else if } \textcircled{2} \wedge ((f = sum \vee min) \wedge (conds(Q_1) \rightarrow a < 0)) \\ \Psi_{Q'_1,Q_1} \wedge b \geq b' & \text{else if } \textcircled{2} \wedge (f = count \vee ((f = sum \vee max) \wedge (conds(Q_1) \rightarrow a > 0))) \\ \Psi_{Q'_1,Q_1} & \text{otherwise} \end{cases}$$

$\textcircled{1}$ $\Psi_{Q'_1,Q_1} \wedge \text{non-grp-pred}(Q_1) \wedge expr(Q_1) \wedge expr(Q'_1) \rightarrow \text{non-grp-pred}(Q'_1)$ $\textcircled{2}$ $\Psi_{Q'_1,Q_1} \wedge \text{non-grp-pred}(Q'_1) \wedge expr(Q'_1) \wedge expr(Q_1) \rightarrow \text{non-grp-pred}(Q_1)$

(b) $\Psi_{Q',Q}$

Figure 7: Rules defining $ge(Q', Q)$ and $\Psi_{Q',Q}$ which are used to test reusability

Subquery $\sigma_{a>10}(R)$ is not contained in $\sigma_{a>30}(R)$, but Q and Q' are equivalent. To avoid failing to determine generalized containment, since it does not hold for a subquery, we do not test generalized containment for selections in ge. Instead we use condition $uconds(Q', Q)$ to test whether all conditions in $pred(Q')$ imply $pred(Q)$:

$$uconds(Q', Q) = \Psi_{Q',Q} \wedge pred(Q') \wedge expr(Q') \wedge expr(Q) \rightarrow pred(Q)$$

For the example shown above this means we test $a = a' \wedge a' = 20 \wedge a' > 10 \rightarrow a = 20 \wedge a > 30$ instead of testing $a = a' \wedge a' > 10 \rightarrow a > 30$ first (which would fail). A similar problem arises when testing whether the input groups for an aggregation are the same for both queries. To avoid failing, because we may not have seen all restrictions for the values of group-by attributes yet, we only check the restrictions on non-group-by attributes enforced by the two queries. Here non-grp-pred(Q) denotes the result of putting $pred(Q)$ into conjunctive normal form and removing all conjuncts that only reference group-by attributes, e.g., given $pred(Q) = a > 10 \wedge g < 5$ where g is group-by attribute, we get non-grp-pred(Q) $= a > 10$. We construct two conditions ① and ② to test whether it is the case that for any group that exists in both $Q'_1(D)$ and $Q_1(D)$, the group for Q'_1 contains a subset of the tuples of the corresponding group for Q_1 (or vice versa). If both ① and ② hold, then Q'_1 and Q_1 produce the same result for every group that exists in both query results. Thus, the aggregation function result produced for these groups by the two queries are equal (we can add $b = b'$ to $\Psi_{Q',Q}$). For the 2nd and 3rd case, we check whether the tuples in a group for Q'_1 is a subset of the tuples for same group in Q_1. If this is the case and are we using *min* or *sum* over negative numbers than then the aggregation function result for Q'_1 is smaller than the one for Q_1. The 3rd case is the symmetric case for *sum* over positive numbers or *max* aggregation.

LEMMA 7. *Let D be a database, Q and Q' be two instances from the same parameterized query $\mathcal{T}$. And $Q = op(Q_1, \ldots, Q_n)$ and $Q' = op(Q'_1, \ldots, Q'_n)$. Then, for all $i \in \{1, \ldots, n\}$,*

$$ge(Q', Q) \Rightarrow ge(Q'_i, Q_i) \quad \Psi_{Q',Q} \Rightarrow \Psi_{Q'_i,Q_i}$$

PROOF. For the rules in fig. 7, $ge(Q', Q)$ is based on $ge(Q'_1, Q_1) \wedge \ldots \wedge ge(Q'_n, Q_n)$. Thus, this lemma holds. □

LEMMA 8. *Given two instances Q and Q' from the same parameterized query $\mathcal{T}$, a database D. Then,*

$$ge(Q', Q) \wedge uconds(Q', Q) \Rightarrow Q'(D) \precsim_{\Psi_{Q',Q}} Q(D)$$

Let $P(t)$ ($P(t')$) denote the provenance of tuple t (t'). For any $\mathcal{M} \subseteq Q'(D) \times Q(D)$ such that $Q'(D) \precsim_{\Psi_{Q',Q}} Q(D)$ holds based on $\mathcal{M}$ we have:

$$ge(Q', Q) \wedge uconds(Q', Q) \Rightarrow \forall (t', t) \in \mathcal{M} : P(t') \subseteq P(t)$$

PROOF. We prove this by induction, that is we need to prove this holds for each subquery of $\mathcal{T}$ between Q and Q'. However, $uconds(Q', Q)$ and $Q'(D) \precsim_{\Psi_{Q,Q}} Q(D)$ might not hold for every subquery, there is a hidden constraint that $\forall (t', t) \in \mathcal{M}$ where $\mathcal{M} \subseteq Q'(D) \times Q(D)$, $P(t) \subseteq P(Q, D)$ and $P(t') \subseteq P(Q', D)$. Thus, what we need to prove in the induction is that for each subquery $\mathcal{T}_{sub}$, let $S = \{t | t \in Q_{sub}(D) \wedge P(t) \subseteq P(Q, D)\}$, $S' = \{t' | t' \in Q'_{sub}(D) \wedge P(t') \subseteq P(Q', D)\}$ and $\mathcal{M} \subseteq S' \times S$ such that $S' \precsim_{\Psi_{Q'_{sub},Q_{sub}}} S$ holds based on $\mathcal{M}$, then

$$ge(Q'_{sub}, Q_{sub}) \wedge uconds(Q', Q) \\ \Rightarrow \\ S' \precsim_{\Psi_{Q'_{sub},Q_{sub}}} S \wedge \forall (t', t) \in \mathcal{M} : P(t') \subseteq P(t).$$

Similar to $P(t)$ and $P(t')$, $P(S)$ ($P(S')$) represents the provenance of set S (S'). And thus, we have $P(Q, D) = P(S)$ and $P(Q', D) = P(S')$.

Base case: When $\mathcal{T}$ is table access operator R, then $Q(D) = Q'(D) = R$. Thus, $Q'(D) \subseteq Q(D)$ which is the special case of $Q'(D) \precsim_{\Psi_{Q',Q}} Q(D)$ where $\Psi_{Q',Q}$ contains equalities on all columns. And $\forall (t', t) \in \mathcal{M}$ where $\mathcal{M} \subseteq Q'(D) \times Q(D)$, we have $P(t) = t$, $P(t') = t'$ and $t = t'$. Hence $P(t') = P(t)$, also $P(t') \subseteq P(t)$.

Inductive step: Assume query Q_n, a sub query of Q, with depth less than or equal to n. Here we use *depth* to represent the levels in the relational algebra format of a query, e.g., Q_1 represents the base table R with depth 1. Assume we have proven that $ge(Q'_n, Q_n) \wedge uconds(Q', Q) \Rightarrow S'_n \precsim_{\Psi_{Q'_n,Q_n}} S_n \wedge \forall (t'_n, t_n) : P(t'_n) \subseteq P(t_n)$, where $(t'_n, t_n) \in \mathcal{M}_n$ and $\mathcal{M}_n \subseteq Q'_n(D) \times Q_n(D)$. Based on the induction hypothesis, we have to prove that the same holds for sub query Q_{n+1} with depth less than or equal to $n + 1$, that is, we need to prove $ge(Q'_{n+1}, Q_{n+1}) \wedge uconds(Q', Q) \Rightarrow S'_{n+1} \precsim_{\Psi_{Q'_{n+1},Q_{n+1}}} S_{n+1} \wedge$

$\forall(t'_{n+1}, t_{n+1}) : P(t'_{n+1}) \subseteq P(t_{n+1})$, where $(t'_{n+1}, t_{n+1}) \in \mathcal{M}_{n+1}$ and $\mathcal{M}_{n+1} \subseteq Q'_{n+1}(D) \times Q_{n+1}(D)$. For notational convenience, we use $op_n(op_{n+1})$ represent any operator of depth n $(n + 1)$, i.e., $Q_{n+1} = op_{n+1}(Q_n)$. Based on lem. 7, $ge(Q'_{n+1}, Q_{n+1}) \Rightarrow ge(Q'_n, Q_n)$. Thus, based on assumption, $ge(Q'_n, Q_n) \wedge uconds(Q', Q) \Rightarrow S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n \wedge \forall(t'_n, t_n) \in \mathcal{M}_n : P(t'_n) \subseteq P(t_n)$. For join operator, we assume the left and right input of op_{n+1} are $Q_{nL}(D)$ and $Q_{nR}(D)$ respectively, then above is hold for both Q_{nL} and Q_{nR}. Thus, S_{nL} (S'_{nL}) is the S (S') in Q_{nL} and S_{nL} (S'_{nL}) is the S (S') in Q_{nL}. Similarly for $\mathcal{M}$ that $\mathcal{M}_{nL} \subseteq Q'_{nL}(D) \times Q_{nL}(D)$ and $\mathcal{M}_{nR} \subseteq Q'_{nR}(D) \times Q_{nR}(D)$. In the following, we consider different op_{n+1}:

For Π_A: Since $\forall(t'_n, t_n) \in \mathcal{M}_n$, there have to be one tuple $t_{n+1} \in Q_{n+1}(D)$ which satisfies $\Pi_A\{t_n\} = \{t_{n+1}\}$ and one tuple $t'_{n+1} \in Q'_{n+1}(D)$ which satisfies $\Pi_A\{t'_n\} = \{t'_{n+1}\}$, and $P(t_n) = P(t_{n+1})$ and $P(t'_n) = P(t'_{n+1})$. Because $S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n$, then $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_n, Q_n}$. In addition, since $P(t'_n) \subseteq P(t_n)$, $P(t'_{n+1}) \subseteq P(t_{n+1})$.

For σ_θ: Since $P(Q, D) = P(S_n)$ and $P(Q', D) = P(S'_n)$, then every tuple $t_n \in S_n$ and every tuple $t'_n \in S'_n$ have to be survived after selection, that is $t_n \in Q_{n+1}(D)$ and $t'_n \in Q'_{n+1}(D)$. Because $S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n$, then $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_n, Q_n}$. In addition, since $P(t'_n) \subseteq P(t_n)$, $t_n = t_{n+1}$ and $t'_n = t'_{n+1}$, $P(t'_{n+1}) \subseteq P(t_{n+1})$.

For δ: Assume $t_{n+1} \in Q_{n+1}(D)$ and $\{t_{n_1}, \ldots, t_{n_k}\} \subseteq Q_n(D)$ such that $\{t_{n+1}\} = \delta\{t_{n_1}, \ldots, t_{n_k}\}$. Similarly, assume $t_{n+1} \in Q'_{n+1}(D)$ and $\{t'_{n_1}, \ldots, t'_{n_m}\} \subseteq Q'_n(D)$ such that $\{t'_{n+1}\} = \delta\{t'_{n_1}, \ldots, t'_{n_k}\}$. Because $S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n$, then $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_n, Q_n}$. Also because $S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n$, $m < k$. And $P(t'_{n_1}) \subseteq P(t_{n_1}), \ldots, P(t'_{n_m}) \subseteq P(t_{n_m})$, then $P(t'_{n+1}) \subseteq P(t_{n+1})$.

$\cup$: $Q_{Ln}(D) \cup Q_{Rn}(D)$ Since $S'_{nL} \precsim_{\Psi_{Q'_{nL}, Q_{nL}}} S_{nL} \wedge \forall(t', t) \in \mathcal{M}_{\backslash\mathcal{L}} : P(t') \subseteq P(t)$ and $S'_{nR} \precsim_{\Psi_{Q'_{nR}, Q_{nR}}} S_{nR} \wedge \forall(t', t) \in \mathcal{M}_{\backslash\mathcal{R}} : P(t') \subseteq P(t)$, after union, the generalized containment will only hold for the common part between $\Psi_{Q'_{nL}, Q_{nL}}$ and $\Psi_{Q'_{nR}, Q_{nR}}$. That is, $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1} \wedge \forall(t', t) \in \mathcal{M}_{\backslash+\infty} : P(t') \subseteq P(t)$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \bigwedge_{a'=a \in \Psi_{Q'_{nL}, Q_{nL}} \wedge a'=a \in \Psi_{Q'_{nR}, Q_{nR}}} a' = a$.

For $\times$: Let $\{t_{n+1}\} = \{t_{nL}\} \times \{t_{nR}\}$ where $t_{nL} \in S_{nL}$ and $t_{nR} \in S_{nR}$. Similarly, let $\{t'_{n+1}\} = \{t'_{nL}\} \times \{t'_{nR}\}$ where $t'_{nL} \in S'_{nL}$ and $t'_{nR} \in S_{nR}$. Since $S'_{nL} \precsim_{\Psi_{Q'_{nL}, Q_{nL}}} S_{nL}$ and $S'_{nR} \precsim_{\Psi_{Q'_{nR}, Q_{nR}}} S_{nR}$, then $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_{nL}, Q_{nL}} \wedge \Psi_{Q'_{nR}, Q_{nR}}$. Since $P(t'_{nL}) \subseteq P(t_{nL})$ and $P(t'_{nR}) \subseteq P(t_{nR})$, then $P(t'_{n+1}) \subseteq P(t_{n+1})$.

For $\bowtie_{a=b}$: Let $\{t_{n+1}\} = \{t_{nL}\} \bowtie_{a=b} \{t_{nR}\}$ where $t_{nL} \in S_{nL}$ and $t_{nR} \in S_{nR}$. Similarly, let $\{t'_{n+1}\} = \{t'_{nL}\} \bowtie_{a=b} \{t'_{nR}\}$ where $t'_{nL} \in S'_{nL}$ and $t'_{nR} \in S_{nR}$. Our join rule in fig. 7 keeps that $t_{nL}.a = t'_{nL}.a$ and $t_{nL}.b = t'_{nR}.b$, thus $(t'_{n+1}, t_{n+1}) \in \mathcal{M}_{n+1}$ and $\mathcal{M}_{n+1} \subseteq Q'_{n+1}(D) \times Q_{n+1}(D)$. That is, $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_{nL}, Q_{nL}} \wedge \Psi_{Q'_{nR}, Q_{nR}}$.Since $P(t'_{nL}) \subseteq P(t_{nL})$ and $P(t'_{nR}) \subseteq P(t_{nR})$, then $P(t'_{n+1}) \subseteq P(t_{n+1})$.

For $\gamma_{f(a) \to b;\, G}$: Assume $t_{n+1} \in Q_{n+1}(D)$ and $\{t_{n+1}\} = \gamma_{f(a) \to b;\, G}(\{t_{n_1}, \ldots, t_{n_m}\})$ where $\{t_{n_1}, \ldots, t_{n_m}\} \subseteq Q_n(D)$, then t_{n+1} either contributes to the result of $Q(D)$ or not. Since $P(Q, D) = P(S_n)$, if yes, $\{t_{n_1}, \ldots, t_{n_m}\} \subseteq S_n$; if not, $\{t_{n_1}, \ldots, t_{n_m}\} \subseteq Q_n(D) - S_n$. That is, if $t_{in} \in S_n$ and $t_{out} \in Q_n(D) - S_n$, then $\forall g \in G : t_{in}.g \neq t_{out}.g$. The same holds for $t'_{n+1} \in Q'_{n+1}(D)$. And because $S'_n \precsim_{\Psi_{Q'_n, Q_n}} S_n$, based on the aggregation rule in fig. 7a, $\forall g \in G : g = g'$, then $S'_{n+1} \precsim_{\Psi_{Q'_{n+1}, Q_{n+1}}} S_{n+1}$ where $\Psi_{Q'_{n+1}, Q_{n+1}} = \Psi_{Q'_n, Q_n}$ and $P(t'_{n+1}) \subseteq P(t_{n+1})$. Now we discuss the relationship between b and b'. Case 1: ① $\wedge$ ② implies that either $Q'_n(D) = Q_n(D)$ or $Q'_n(D)$ and $Q_n(D)$ different on the predicates of containing **GROUP BY** attributes. If all of the operators in $\mathcal{T}_n$ are monotone operators, then for the tuples in D with the same values on **GROUP BY** attributes, either all of them will survive together until the aggregation (op_{n+1}) or all of them are filtered out together by the predicates before the aggregation. Then each common group of op_{n+1} over S'_n and S_n are derived from the same tuples from D, thus $b = b'$. If $\mathcal{T}_n$ contains aggregations, then the **GROUP BY** attributes of each of these aggregations should include the the **GROUP BY** attributes of op_{n+1}, then the case discussed above still holds. Thus $b = b'$. Case 2: since ② implies that for each of these groups mentioned above, the group in $Q'_n(D)$ contains less tuples compared with the group in $Q_n(D)$, and thus $b \leq b'$ if $((f = sum \vee min) \wedge (conds(Q_1) \to a < 0))$. Case 3: if $(f = count \vee ((f = sum \vee max) \wedge (conds(Q_1) \to a > 0)))$, then $b \geq b'$. Case 4: Otherwise, we let relationship between b and b' be undecidable.

□

LEMMA 9. *Given Q and Q' two instances of a parameterized query $\mathcal{T}$, a database D, consider $\mathcal{M} \subseteq Q'(D) \times Q(D)$ such that $Q'(D) \precsim_{\Psi_{Q',Q}} Q(D)$ holds based on $\mathcal{M}$, then*

$$Q'(D) \precsim_{\Psi_{Q,Q'}} Q(D) \wedge \forall(t', t) \in \mathcal{M} : P(t') \subseteq P(t)$$
$$\Rightarrow P(Q', D) \subseteq P(Q, D)$$

PROOF. Since $Q'(D) \precsim_{\Psi_{Q,Q'}} Q(D)$, that is $\forall t' \in Q'(D) : \exists t \in Q(D) : \mathcal{M}(t', t)$. Because $P(t') \subseteq P(t)$, $P(Q', D) \subseteq P(Q, D)$. □

Thus, based on the discussion by now, we could infer in a reverse order that whether the provenance sketches of Q could answer Q'.

THEOREM 5. *Let Q and Q' be two instances of a parameterized query $\mathcal{T}$, D be a database, and $\mathcal{PS}$ a set of safe provenance sketches of Q with respect to D.*

$$ge(Q', Q) \wedge uconds(Q', Q) \Rightarrow \mathcal{PS} \textbf{ is safe for } Q' \textbf{ and } D$$

PROOF. In turn in lem. 8 and lem. 9 we get $P(Q', D) \subseteq P(Q, D)$. Let $\mathcal{PS}_{ac}$ and $\mathcal{PS}'_{ac}$ be the accurate provenance sketches of Q and Q' with respect to D. They are the same type with $\mathcal{PS}$. From lem. 6, $P(Q', D) \subseteq P(Q, D) \Rightarrow D_{\mathcal{PS}_{ac}} \supseteq D_{\mathcal{PS}'_{ac}}$. Because $D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}_{ac}}$, $D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}'_{ac}}$. At last, in turn in lem. 5 and lem. 4 that if $D_{\mathcal{PS}}$ is safe to answer Q, then it is also safe to answer Q'. □

EXAMPLE 10. *Consider the parameterized query $\mathcal{T} = \sigma_{cnt>\$2}(\gamma_{state;count(*)\to cnt}(\sigma_{popden>\$1}(cities)))$. This query returns states that have more than $2 cities with a population density of at least $1. Assume Q and Q' are two instances of $\mathcal{T}$ with parameters binding* $(100, 10)$ *and* $(100, 15)$ *for* $(\$1, \$2)$ *respectively. We use Q_{agg} and Q'_{agg} to denote the subqueries rooted at the aggregation operator. To determine whether a set of sketches $\mathcal{PS}$ for Q can be used to*

```
SELECT state, count(city) AS cntcity
FROM cities
GROUP BY state
WHERE popden > $1
HAVING cntcity > $2
```

	Q	Q'
$1	100	100
$2	10	15

Figure 8: Parameterized Query $\mathcal{T}$ and instances Q and Q'

answer Q', we construct the conditions shown below. We use p, c, and s to denote popden, city, and and state, respectively.

$$pred(Q) = p > 100 \wedge cnt > 10 \quad pred(Q') = p' > 100 \wedge cnt' > 15$$

$$\Psi_{Q',Q} = p = p' \wedge c = c' \wedge s = s' \wedge cnt = cnt'$$

Since this query does not contain any projections, $expr(Q)$ and $expr(Q')$ are empty. The condition $ge(Q', Q)$ constructed for this query test the relationship between group-by attributes in the inputs of the aggregation subqueries Q_{agg} and $Q_{agg'}$. Since $\Psi_{Q'_{agg},Q_{agg}}$ contains $s = s'$, $ge(Q', Q)$ holds. Furthermore, both ① and ② hold and, thus, we add $cnt = cnt'$ to $\Psi_{Q',Q}$. Finally, $uconds(Q', Q)$ tests

$$\Psi_{Q',Q} \wedge pred(Q') \wedge expr(Q') \wedge expr(Q) \rightarrow pred(Q)$$

Substituting the conditions shown above we get $p = p' \wedge cnt = cnt' \wedge p > 100 \wedge cnt' > 15 \wedge p' > 100 \wedge cnt > 10$. Since this condition holds for all possible values of the variables in the formula (recall that free variables are assumed to be universally quantified), we can use $\mathcal{PS}$ to answer Q'.

10 SELF-TUNING

To be able to use PBDS to optimize workloads consisting of multiple instances of one or more parameterized queries, we design a simple self-tuning strategy. We leave a detailed study of self-tuning and more complex strategies to future work. We designed strategies to decide for each incoming query whether we will capture a sketch, use a previously captured sketch, or just execute the query without any instrumentation. We use our safety tests to determine which attributes are safe for a parameterized query and the method described in Sec. 9 to determine whether one of the sketches we have captured can be used to answer an incoming query.

Self-tuning Strategies. *Eager strategy*: We keep track of sketches we have captured using a map between values of the query parameters and sketches. Since sketches are not effective for non-selective queries, we estimate the selectivity of queries and if it is above a threshold (75% in our experiments), we execute the query without using a sketch. Otherwise, we employ the techniques from Sec. 9 to check whether any of the sketches we have captured so far can be used. reusability check rules. If this is the case, we instrument the query to use this sketch. If no such sketch exists, then we record what sketch could have been used for the query. To avoid paying overhead for sketches that are rarely used, we only create a new sketch once we have accumulated enough evidence that the sketch is needed (the number of times it could have been used is above a threshold). We call this the *adaptive strategy*.

11 EXPERIMENTS

All experiments were run on a machine with 2 x 3.3Ghz AMD Opteron 4238 CPUs (12 cores) and 128GB RAM running Ubuntu 18.04 (linux kernel 4.15.0). We use Postgres 11.4, MonetDB 11.33.11 and DB-X (name omitted due license restrictions). In all experiments, we determined what attributes are safe using the techniques from Sec. 8. For experiments measuring the end-to-end performance of PBDS, the cost of safety and reuse checks is included in the runtime.

11.1 Workloads and Datasets

TPC-H. We use the TPC-H [1] benchmark at SF1 (~ 1GB) and SF10 (~ 10GB) to evaluate performance.

Crimes. This dataset records crimes reported in Chicago (https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2). It contains ~6.7M tuples.We use two queries: `C-Q1`: Compute the 5 areas with the most crimes. `C-Q2`: Return the number of blocks where more than 10000 crimes took place.

Movie Ratings. The MovieLens dataset (https://grouplens.org/datasets/movielens) contains a movie relation (~27k movies) and a ratings relation (~20m ratings). We use three queries: `M-Q1`: Compute the 10 movies with the most ratings. `M-Q2`: Return the number of movies with more than 63,300 ratings. `M-Q3`: Return the 10 most popular movies where popularity is defined as the weighted sum of the number of ratings of a movie and the number of times it has been tagged.

Stack Overflow (SFO). This is an archive of content from https://www.kaggle.com/stackoverflow/stackoverflow. It consists of relations: users (~12.5m rows), badges (~35.9m rows), comments (~75.9m rows) and posts (~48.5m rows). We use ten real queries from or modified from https://data.stackexchange.com/stackoverflow/queries: `S-Q1`: The 10 users with the most number of posts. `S-Q2`: Owners of the 10 most favored posts. `S-Q3`: The 10 users that authored the most comments. `S-Q4`: The 10 users with the most badges. `S-Q5`: Users who did post between 47945 and 52973 comments. `S-Q6`: Return the top 100 users with the highest average answer score excluding community wiki or users with less than 10 answers. `S-Q7`: Return the average upvotes for tags answered by > 1000 users. `S-Q8`: Return the most popular tags in May 2010. `S-Q9`: Return the top 100 controversial posts ($\#downvotes > 0.5 \cdot \#upvotes$). `S-Q10`: Return the top 10 downvoted users. `S-Q11`: Return the 10 users with the most answers.

11.2 Capture Optimizations

We first evaluate the effectiveness of the optimization for provenance capture presented in Sec. 5.3. Fig. 12a and 12b show capture runtime varying $|F|$, i.e., the number of fragments of the partition based on which we are creating the sketch. We use PSi to denote a partition with i fragments.

Creating Singleton Sketches. We considered two approaches for creating singleton sketches for tuples by determining which fragment of a range-partition the tuple belongs to. Either the membership of the tuple is tested using a list of *case* expressions or we use a UDF to perform *binary search* over the ranges of the partition. We use the crimes dataset and Postgres in this experiment. As expected, *binary search* significantly outperforms *case* for larger number of fragments, e.g., about 2 orders of magnitude for PS10K.

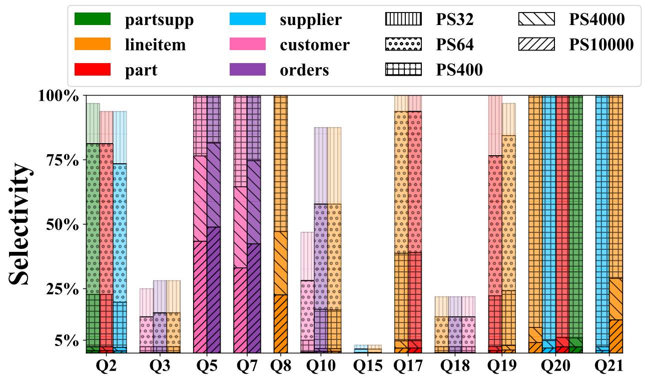
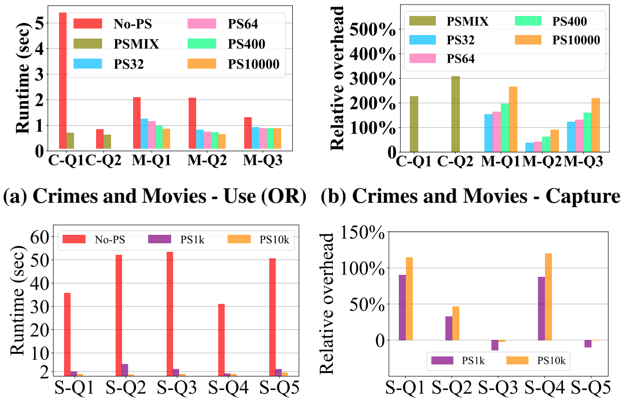


Figure 9: Selectivity of range-partition provenance sketches for the TPC-H 1GB.



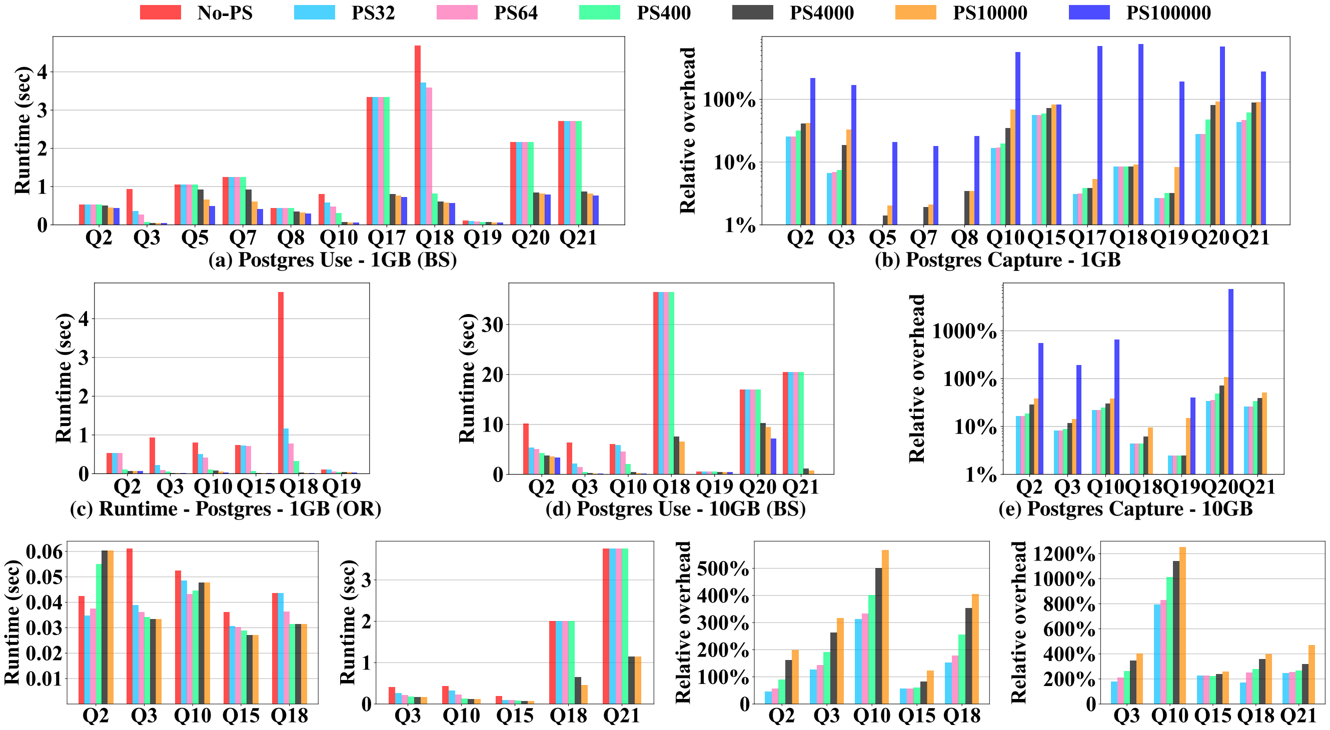
(a) Crimes and Movies - Use (OR)

(b) Crimes and Movies - Capture

(c) SOF - Use (BS)

(d) SOF - Capture

Figure 10: Real world data



(a) Postgres Use - 1GB (BS)

(b) Postgres Capture - 1GB

(c) Runtime - Postgres - 1GB (OR)

(d) Postgres Use - 10GB (BS)

(e) Postgres Capture - 10GB

(f) MonetDB Use - 1GB (OR)

(g) MonetDB Use - 10GB (OR)

(h) MonetDB Capture - 1GB

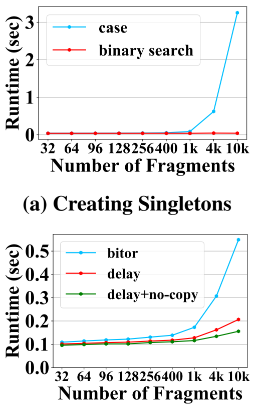
(i) MonetDB Capture - 10GB

Figure 11: Performance of provenance sketch capture and use for TPC-H queries.

Merging Sketches.: For operators like aggregation we need to union sketches which corresponds to computing the bitwise-or of sketches. Recall that we presented two optimizations for the operation in Sec. 5.3: *delay* and *No-copy*. For this experiment, we union the singleton sketches for all tuples from the movie ratings datasets.The results are shown in Fig. 12b. The *delay* optimization significantly improves performance for larger number of fragments, e.g., from ~0.5 seconds to ~0.2 seconds for PS10K. *No-copy* further improves this to ~0.16 seconds. Thus, we enable these optimizations for all remaining experiments.

11.3 TPC-H

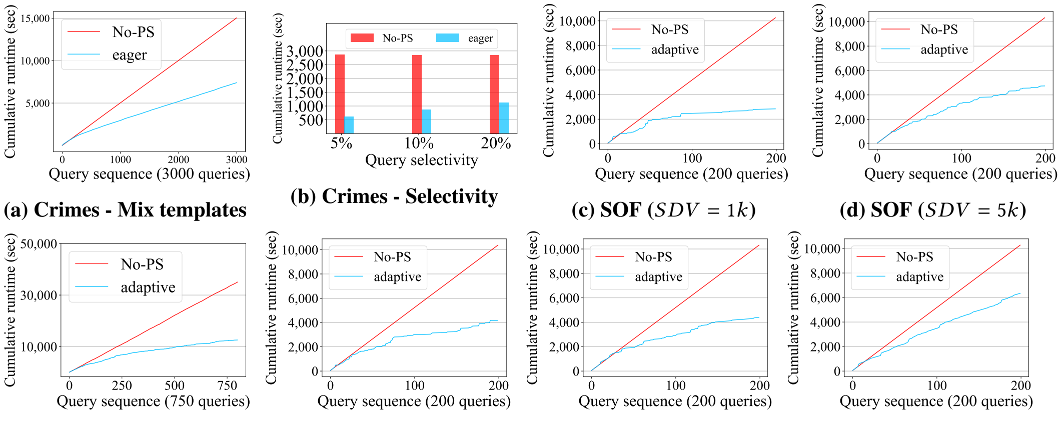
Because of the TPC-H's artificial data distribution, this stresses our approach since there are essentially no meaningful correlations that we can exploit. As explained in Sec. 6 , we use equi-depth histograms maintained as statistics by the DBMS to determine the partition ranges for sketches. We generate sketches on primary key attributes (PK). However, for cases where the PK is unsafe, we build sketches over the query's group-by attributes. PK attributes have the advantage that both Postgres and MonetDB automatically build indexes on PK columns. We first evaluated how the number of fragments of a partition affect the selectively of sketches (the fraction



(a) Creating Singletons

(b) Merging Sketches

Figure 12: Optimizations



(a) Crimes - Mix templates

(b) Crimes - Selectivity

(c) SOF ($SDV = 1k$)

(d) SOF ($SDV = 5k$)

(e) SOF - Mix templates

(f) SOF ($sel = 0.7\%$)

(g) SOF ($sel = 2\%$)

(h) SOF ($sel = 5\%$)

Figure 13: End-to-end Experiments

	1GB				10GB			
Query	No-PS	PS4000	PS10000	PS100000	No-PS	PS4000	PS10000	PS100000
Q2	[1, 10)		[10, 73)	[73, ∞)	[1, 3)	[3, 6)	[6, 215)	[215, ∞)
Q3	[1, 2)	[2, ∞)			[1, 2)	[2, 17)	[17, 1629)	[1629, ∞)
Q5	[1, 3)			[3, ∞)	[redacted]	[redacted]	[redacted]	[redacted]
Q7	[1, 2)			[2, ∞)	[redacted]	[redacted]	[redacted]	[redacted]
Q8	[1, 4)		[4, 5)	[5, ∞)	[redacted]	[redacted]	[redacted]	[redacted]
Q10	[1, 2)	[2, 46)	[46, 667)	[667, ∞)	[1, 2)	[2, 3)	[3, 462)	[462, ∞)
Q17	[1, 2)		[2, 799)	[799, ∞)	[redacted]	[redacted]	[redacted]	[redacted]
Q18	[1, 2)	[2, 3)	[3, 1968)	[1968, ∞)	[1, 2)		[2, ∞)	
Q19	[1, 3)		[3, 195)	[195, ∞)	[1, 7)		[7, 8)	[8, ∞)
Q20	[1, 3)	[3, 11)	[11, 387)	[387, ∞)	[1, 5)	[5, 8)	[8, 540)	[540, ∞)
Q21	[1, 15)		[15, 507)	[507, ∞)	[1, 6)	[6, 26)	[26, ∞)	

Figure 14: Optimal #fragments varying #repetitions

of input data covered by the sketch). We discuss these results in detail below. For queries that are omitted in the following either the provenance is too large for these queries to benefit from PBDS (e.g., Q1's provenance is over 95% of its input) or the query's selection conditions leave no room for improvement.

Provenance Sketch Selectivity. Fig. 9 shows the percentage of data covered by provenance sketches for each relation accessed by a query when varying the number of fragments from 32 to 10000. We use colors to denote relations and patterns to denote number of fragments. For consistency we generate provenance sketches on the primary key attributes of a relation. However, for cases where using the PK would be unsafe (see Sec. 8) build the sketch over the query's group-by attributes. For queries that are not shown here either the provenance is already too large for this queries to benefit from PBDS or the query's selection conditions are already restrictive leaving no further room for improvement. For example, query Q1's provenance consists of over 95% of its input (the lineitem table). Note that about half of the TPC-H queries are quite selective in terms of provenance and this selectivity can be exploited by sketches even when using only a moderate number of fragments. For many queries we already achieve selectivities of a few percent for PS4000.

Postgres - Capture & Reuse. Next, we evaluate whether these input size reductions lead to significant performance improvements. Fig. 11a and 11d show the runtime of TPC-H queries using captured sketches (*PS*) and without PBDS (*No-PS*). We created zone maps (called brin indexes in Postgres) for all tables. Furthermore, we create indexes on PK and FK columns. Note that PK indexes are created automatically by the system. Unless stated otherwise, queries apply the binary search (*BS*) method to test whether a tuple belongs to a sketch (Sec. 7).

Fig. 11a shows runtimes for SF1. Q3 is a top-10 query that returns the 10 orders with the highest revenue. It is highly selective on the PK of orders and customer (at most 10 customers have submitted these orders). Since we use equi-depth histograms to determine partition ranges, each fragment contains approximately the same number of rows. Thus, the runtime of the query is roughly linear in the number of rows contained in the 10 fragments of the sketch, e.g., $\sim \frac{1}{40}$ the runtime without PBDS for PS400. We observe similar behavior for Q10 and Q18 which are top-20 and top-100 queries, respectively. The result for Q19 demonstrates that PBDS can sometimes unearth additional ways to exploit selection conditions that the DBMS was unable to detect. As shown in Fig. 9, for queries Q5, Q7, Q8, Q20 and Q21 we need larger numbers of fragments to be able to benefit from PBDS. This trend is also reflected in query performance Fig. 11a. While Q2 and Q17 have selective sketches, their selection conditions are quite restrictive leaving little room for improvement. Fig. 11d shows runtimes for SF10. Observe that the runtime of queries Q2, Q3, Q10, Q20 and Q21 exhibit similar behavior as for SF1.

Binary search is typically more efficient when the number of fragments in the provenance sketch is large. However, for very selective provenance sketches, using a B-tree index will be more efficient. Fig. 11c shows the runtime of queries with selective provenance sketches over a SF1 instance when translating the sketch into a disjunctive condition (*OR*). We only show queries whose runtime is improved compared to using binary search. The most significant improvements are for Q15 which did not benefit from PBDS for binary search and Q2 whose runtime is reduced to 0.1 seconds for PS400 and 0.066 seconds for PS10000.

Fig. 11b and 11e show the overhead of capturing sketches relative to executing the queries without any instrumentation for SF1 and SF10. For some queries the overhead is less than 20% while it is always less than 100% for partition sizes up to 10000 fragment. The overhead increases slightly in the number of fragments since larger number of fragments result in larger bitvectors. Here we benefit from using binary search instead of a linear sequence of **CASE** expressions. For 100000 fragments the overhead is typically between 20% and 700% with an outlier (Q20) for the 10GB database which has ~7500% overhead. For Q18 and Q21, the capture query did not finish in the time we allocated for each experiment.

Amortizing Capture Cost. We now analyze whether the overhead of capture can be amortized by using provenance sketches. Fig. 14 shows for each TPC-H query and a given number of repetitions (n_{runs}) of this query, the partition size (if any) that minimizes total query execution cost. Note that these numbers are for Postgres. For each partition size (and no partitioning) we show the interval of repetitions for which this option is optimal. For example, consider query Q10 and the 1GB instance, [1,2) in column No-PS means that if we only need to run the query once, then the optimal choice is to not create any provenance sketch. Let C_{No-PS}, C_{cap} and C_{reuse} represent the cost of running the query, capturing the provenance sketch for the query, and running the instrumented query which uses the sketch, respectively. Then the cost of evaluating the query $runs$ times without PBDS is $C_{No-PS} * n_{runs}$. When using a sketch we have to create the sketch and then evaluate the query n_{runs} times using the sketch: $C_{cap} + C_{use} * n_{runs}$. Based on these formulas we can determine which option is optimal for n_{runs} repetitions. We do not show PS32, PS64, and PS400 since these options are dominated by other options for all values of n_{runs}. Note that use of provenance sketches for PBDS often results in performance improvements of several orders of magnitude. Thus, the overhead of capturing a sketch is often amortized by using the sketch only once or twice. Cells which are blacked out are queries for which PBDS is not beneficial for this dataset size.

11.4 MonetDB

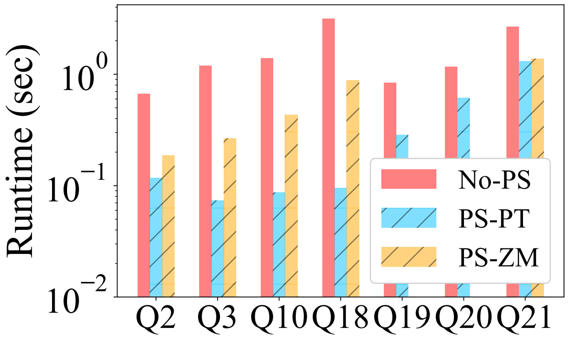
We also evaluate PBDS on MonetDB to test our approach on an operator-at-a-time columnar main-memory system without indexes that is optimized for minimizing cost per tuple. Fig. 11f and 11g show the runtime for using sketches. Even though MonetDB supports database cracking [53] and column imprints [86] (a technique similar to zone maps), the implementation of these techniques turned out to not be beneficial for PBDS. The histograms used by column imprints as statistics for skipping are too-coarse grained for PBDS and database cracking evaluates each range condition of a selection condition separately instead of determining statically what fragments of the cracked column have to be read. We explain in detail why this is the case below. Nonetheless, PBDS is still beneficial for several queries. However, for 1GB the overhead of evaluating **WHERE** clause conditions sometimes outweight the benefits of reducing data size (Q2 and Q10). Fig. 11h and 11i shows the relative overhead of sketch capture (similar trends as for Postgres). We omit PS100000 since it did not result in additional improvement.

Physical Design in MonetDB. The performance of provenance sketch use depends on how efficient we can skip data that does not belong to the sketch. In MonetDB, there are hash indexes that the system builds for primary key columns, database cracking[53] (created with **CREATE ORDERED INDEX**), range partitioning, and column imprints [86] (created with **CREATE** `IMPRINTS` **INDEX**). Hash indexes are obviously not a great fit for range queries.

Database cracking [53] incrementally divides a column into partitions such that each partition contains a particular range of values matching the selection condition of a query. Data is not sorted within a partition. A small memory-resident binary tree indexes the fragments and is used to determine what fragments have to be read to answer a query. On paper, database cracking should work well with provenance sketch use. However, we experimentally evaluated the performance of MonetDB with cracking activated on columns used in provenance sketches on the 10GB TPC-H workload and performance was worse than without cracking. We believe that this likely due to the fact that for fine-granular provenance sketches, a column is split into too many small fragments. This increases the cost during query optimization to determine which of the fragments are relevant. However, more importantly, each range condition of the selection condition corresponding to a provenance sketch is evaluated as a several MAL instruction (MAL is MonetDB's internal VM language that expresses query plans as array operations). Such plans do not exploit the fast batch processing capabilities of MonetDB. Furthermore, MonetDB does not eliminate selection conditions that are guaranteed to evaluate to true on every row of a fragment.

Column imprints [86] supported in MonetDB are similar to zone maps. A column is divided into chunks of one cache-line each and summary statistics are recorded for each chunk. The imprint of a chunk records for each bucket of a histogram over the column's values whether the chunk contains at least one value that falls within this bucket. This information is stored as a (compressed) bitvector. Given a selection condition, MonetDB evaluates for each chunk whether the selection condition may evaluate to true for any of the buckets for which there are rows in the chunk. If that is not the case, then the chunk can be skipped. Zonemaps typically work well for PDBS. Column imprints are less effective. There are two reasons for that. First off, the histograms used for column imprints have a low number of buckets, because the chunks are quite small and, thus, a larger number of buckets would result in too much storage overhead. For fine-grained provenance sketches, these histograms are too coarse-grained and, thus, do not result in much skipping potential. Furthermore, MonetDB generates a separate scan for each disjunction in the selection condition encoding a provenance sketch which again results in too much overhead in terms of operator handling and also in terms of checking whether a chunk has to be read.

As we will show in Sec. 11.5, horizontal partitioning can be quite effective for PBDS. While MonetDB supports partitioning, the plans generated for evaluating the selection conditions for sketch use are highly inefficient, because they consist of a large number of operators. As an example for this issue consider the query shown below that filters data based on a provenance sketch that contains 52 fragments (resulting in 52 range conditions). The aggregations are chosen to force the system to access several columns of the table. When partitioning the `lineitem` table into 100 fragments, the plan consists of over ~ 40,000 MAL instructions (the array-operation language used by MonetDB to express query execution plans). When partitioning the table into 1000 fragments, this increases to ~ 430,000 MAL instructions. The runtime of this plan over a SF10 TPC-H instance was about 48 seconds while running this query over a table that is not partitioned takes only $sim70$ miliseconds.



(a) DB-X Use - 10GB (OR)

```
SELECT count(*), min(l_partkey * 2), sum(l_linenumber / 100)
  FROM lineitem
 WHERE (l_orderkey >= 0 AND l_orderkey < 600000) OR
       (l_orderkey >= 600000 AND l_orderkey < 1200000) OR
       (l_orderkey >= 1200000 AND l_orderkey < 1800000) OR
       ...
       (l_orderkey >= 59400000 AND l_orderkey < 60000000);
```

In summary, MonetDB supports index structures and range partitioning. In principle these techniques should be quite effective for PBDS. However, because of how these structures are implemented in MonetDB, these techniques cannot be utilized effectively for PBDS. We believe that some changes to the implementation of these data structures and how are utilized by the system (e.g., creating a single scan for reading all relevant fragments of a cracked column and eliminating selection conditions that are guaranteed to hold on a fragment instead of checking them for each value) would greatly improve PBDS performance on MonetDB. To demonstrate that our hypothesis that sketches are useful for in-memory data processing, we have also run the TPC-H 10GB query workload (using sketches) on a commercial system DB-X (name excluded due to licensing restrictions). DB-X supports in-memory columnar caching of data, supports zone maps, indexes, and partitioning. We present these results in Sec. 11.5.

11.5 TPC-H DB-X

We also evaluated PBDS on the cloud deployment of *DB-X*, a commercial DBMS with support for columnar storage. We measured the performance of sketch use for the same TPC-H queries as for Postgres for SF10. We use a VM with 16 shared CPUs and did evaluate our approach for a database with physically range-partitioned tables (*PS-PT*) and for tables with zone maps (*PS-ZM*). The results are shown in Fig. 15a. PS-ZM outperforms No-PS by a factor of ~ 2 to ~ 4.6. PS-PT is always the best choice and improves performance by a factor of ~ 2 to ~ 37. The reason for the superior performance of PT is that zonemap cannot utilize binary search when determining whether a chunk can be skipped. This is also why Q19 and Q20 runtimes are omitted for PS-ZM: the provenance sketches for these queries are quite large, leading to runtimes that are slower than No-PS, because the selection condition for skipping has to be checked against the min/max values for each chunk of data. Sketch capture is inefficient in DB-X, because our implementation of binary search as a UDF suffers from the high overhead of UDF calls in DB-X (numbers of omitted).

11.6 Real World Datasets

Crimes. C-Q1 is a top-5 query grouping on geographical attributes while C-Q2 is two-level aggregation where the inner aggregation groups on geographical attribute *block*. We consider provenance sketches over a combination of all group-by attributes (denoted as PSMIX). Since these are strongly correlated geographical attributes with a low number of distinct values, i.e., a one-to-one correspondence between fragments and group-by values. We show the runtime of PBDS and capture in Fig. 10a and 10b, respectively. PBDS improves performance by 88.5% for C-Q1 and 30.3% for C-Q2. The capture overhead is larger in this experiment than for the TPC-H queries since these queries do not use any selection conditions and thus a singleton sketch has to be produced for every input row.

Movies. Similar to the crime dataset we build provenance sketches over the group-by attributes.The main difference is that the number of distinct values in the group-by attribute (`movieid`) is quite large. The runtime of M-Q1, M-Q2 and M-Q3 is improved by 61% , 72.2% and 35%, respectively for PS10000. The capture overhead ranges between a factor of ~ 0.37 and ~ 3.08 (none of these queries contain any selection conditions).

Stack Overflow (SOF). Compared with crimes and movies dataset, this is a very large dataset, thus we only consider 1000 and 10000 fragments. Fig. 10c shows that PBDS is quite effective improving query performance by 96.9% to 98.85% for PS10000. The capture overhead ranges between a factor of ~ −0.14 and ~ 1.2 (Fig. 10d). The negative overhead is caused by Postgres choosing parallel pre-aggregation for the capture query, but not for the No-PS query. We also present results for several real world datasets.

11.7 End-to-end Experiment

We now evaluate PBDS in a self-tuning setting on workloads that consist of multiple instances of one or more parameterized queries using the techniques described in Sec. 10. Note that the runtimes reported here include the runtime of safety and reuse checking and the cost of creating sketches. We generate each template-based query instance based on following steps: 1) rolling a dice to decide which template to use; 2) random choosing parameter values (the values satisfy normal distribution per template); 3) for interval paramaters, e.g., $a > \$1$ and $a < \$2$, we do step 2) two times, one for start point (\$1), one for the interval size n, then $\$2 = \$1 + n$. We evaluate the performance of our strategy varying query selectivity (*sel*) and the parameters of the normal distribution used to determine constants used in selection conditions. For this experiment we modified queries introduced in Sec. 11.1 by changing **LIMIT** to a **HAVING**.

Crimes. In Fig. 13a, we mix four templates and each one contains up to five parameters. Recall *eager* method creates sketch if no existing sketch could be used which might spend much time in the early stage to creating large number of sketches, thus we start to gain from the 133_{th} queries. However, these costs would be amortized with following queries, e.g., 50.8% performance improvement until the 3000_{th} query. Furthermore, we learn the influence by varying the query selectivity. In Fig. 13b, we generate different selectivity queries from single template. As we imagined that high selectivity results in a lower performance improvement, however, we still improve performance by 60% at 20% selectivity.

Stack Overflow (SOF). Fig. 13e shows the result of a workload over the stack overflow dataset with three query templates (SQL code shown in Sec. 11.9). We set $sel = 1\%$. Since our *adaptive* strategy (see Sec. 10) delays capturing sketches and we pay for creating sketches, there is a delay before we see benefits. The benefits of using sketches accumulate over time and *adaptive* outperforms *No-PS* by $\sim 3x$ with respect to total workload execution time (800 queries). We also evaluated how query selectivity affects performance. Fig. 13f to 13h show results varying the average query selectivity (0.7%, 2% and 5%). For this experiment we use a single query template. As expected we benefit less for higher selectivites. We also varied the standard deviation (*SDV*) of the normal distribution we use to determine parameter values (1000 and 5000) and fix selectivity to 1%. As expected (Fig. 13c and 13d), we accumulate benefits faster when parameter values are more clustered (*SDV*=1000), because a smaller number of sketches is sufficient for covering most queries.

Safety and Reuse Check Overhead. We separately measured the overhead of safety and reuse checks (both are $\sim$ 20 ms per check). *Safety checks*: If there are n sets of columns we want to check, then the total cost is $0.02 * n$ seconds. Since we only need to evaluate safety once per query template, this cost is negligible. *Reusability check*: Given k templates and m sketches for the each template, we have to test which template a query corresponds to. This takes about 0.05 ms per template. Then we need to check for each sketch we have created for the query's template whether it can be used to answer the query. Thus, the total time requires to find a sketch to use is $k \cdot 0.00005 + 0.02 \cdot m$ seconds.

11.8 Provenance Sketches vs Materialized Views

We now compare the performance and space usage of PBDS (*PS*) against query answering with materialized views (*MV*). Furthermore, we also consider combining these two methods by building provenance sketches on-top of MVs and/or using MVs for some parts of a query and PBDS for others. For PBDS, the performance is affected by the provenance sketches size ratio (PSSR = number of fragments in provenance sketches / number of fragments in total) and the cost of filtering data that does not belong to the sketch. The performance of materialized views is affected by the materialized view size (MVS) and the cost of the remaining parts of the query. We start with a synthetic dataset with 40M rows where the above factors can be fully controlled. Afterwards, we verify the results using real datasets and queries and for a self-tuning setting over workloads over a single parameterized query.

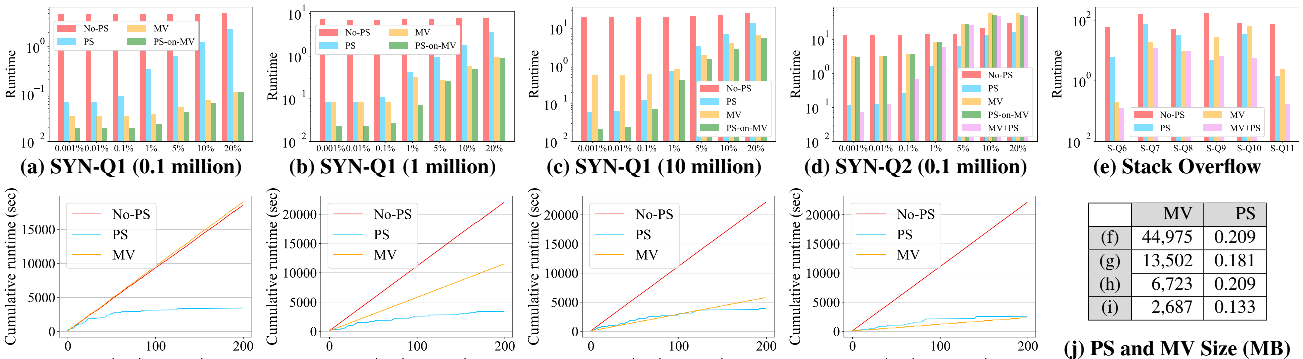
Synthetic Datasets. We use a group-by aggregation with **HAVING** (template SYN-Q1) which is beneficial for MVs. We materialize the aggregation result. We control the number of groups by choosing the group-by column (MVS). We vary the **HAVING** condition to control the query result size which determines PSSR. Fig. 16a to 16c show the runtime for answering SYN-Q1 for MVS from 0.1 (Fig. 16a) to 10 million (Fig. 16c) and PSSR from 0.001% to 20%. PS is more effective for SYN-Q1 for lower selectivities, because the sketch will be small and most of the MV's data is irrelevant for the query. For MVS 0.1 million (fig. 16a), MV outperforms PS since we only need to evaluate the having condition on the small MV. However, when increasing MVS to 1 million (fig. 16b), for selective having conditions (small PSSR like 0.001% and 0.01%), the two techniques exhibit similar performance. For MVS 10 million (fig. 16c), PS is always better. Build a sketch on the MV (PS-on-MV) outperforms both methods for this template. Fig. 16d shows the runtime for template SYN-Q2 which joins the result of a group-by aggregation with **HAVING** with another table. For MV, we did materialize the aggregation result. Even for MVS 0.1 million (Fig. 16d), PS significantly outperforms MV for all selectivities, because PS can filter both inputs of the join. We consider two options for combining PBDS and MV: *PS-on-MV* builds a sketch on the MV. *PS+MV* uses the MV and uses a sketch for the joined table. Based on the query's selectivity query and the MVS, applying sketches on-top of the MV may be worse than just using the MV. For SYN-Q2, PS outperforms hybrid options for all settings with the exception of PSSR 0.001% and 0.01% where PS+MV is slightly faster.

Stack Overflow Data. To verify whether the results for synthetic dataset translate to real world settings, we compared the two techniques on the Stack Overflow dataset using real queries posed by stackoverflow users. We choose six representative queries. The results are shown in Fig. 16e. Note here, we only materialize aggregation results (we also treat **DISTINCT** as an aggregation). For queries with more than one level of aggregation, we choose to materialize the innermost aggregation. Query S-Q6 is an aggregation over a join of two tables returning the top-k aggregation results. Directly running the top-k operation on the materialized aggregation (MV) with better performance. Query S-Q7 first computes a **DISTINCT** over a join of four tables, then joins the result of this subquery with two tables, and finally a group-by aggregation with **HAVING**. Since PSSR is high for this query, using provenance sketches are less effective. Query S-Q8 is a query joins the result of two separate aggregations. Since the aggregation results are relatively small MV outperforms PS. Queries S-Q9, S-Q10 and S-Q11 have a similarly structure as S-Q7, consisting of an aggregation and a join afterwards followed by a top-k operator or an aggregation and top-k. PS outperforms since for the top-k query PSSR is relatively low and we can use sketches to filter joined tables early on. Notably, the hybrid approach (PS+MV) outperforms both PS and MV in almost all cases.

End-to-end Workloads. Next, we compare MV and PS in a self-tuning setting over the Stack Overflow dataset. We apply the *eager strategy* (Sec. 10) and use the following query template:

```
SELECT count(*) AS cnt, u_id, u_displayname FROM comments, users
WHERE c_creationdate >= $1 AND c_creationdate < $2 AND c_userid=u_id
GROUP BY u_id, u_displayname HAVING count(*) >= $3
```

We ran 200 instances of this template using different options for setting the $1, $2 and $3 parameters. *Strategies*: Consider a specific instance of the template for $\$1 = c_1, \$2 = c_2, \$3 = c_3$. The result of the aggregation depends on $1 and $2 and, thus, a separate view has to be created for each such setting. Checking reusability of an existing sketch for this query corresponds to checking the containment of intervals $[c_1, c_2)$ and $[c_3, \infty)$ in the intervals for the sketch. If no such sketch exists, then we capture a new sketch. In either case, we then evaluate the query using the new or an existing sketch. *Results*: Fig. 16 shows the costs. Both methods are highly efficient if an existing sketch/view can be reused. Thus, the main cost



(a) SYN-Q1 (0.1 million) (b) SYN-Q1 (1 million) (c) SYN-Q1 (10 million) (d) SYN-Q2 (0.1 million) (e) Stack Overflow

(f) 200 ($1,$2), 1 ($3) (g) 100 ($1,$2), 2 ($3) (h) 50 ($1,$2), 4 ($3) (i) 20 ($1,$2), 10 ($3)

	MV	PS
(f)	44,975	0.209
(g)	13,502	0.181
(h)	6,723	0.209
(i)	2,687	0.133

(j) PS and MV Size (MB)

Figure 16: Comparing PBDS (PS) against materialized views (MV): (a)-(e) show performance of using sketches / views for query answering for two queries over synthetic data and Stack Overflow (e); (f)-(h) show the end-to-end performance of both methods for a workload consisting of multiple instances of a query template; (j) shows the storage requirements for sketches and MVs for (f)-(i).

is creating MVs and sketches. Since the number of MVs and sketches that need to be created are affected by the number of the distinct settings of the inner selection ($1 and $2) and the outer selection ($3), we control for these settings. Larger number of distinct ($1, $2) pairs are disadvantageous for MV (a view is created for each ($1, $2) pair). PS outperforms MV for these workloads (except for 20 ($1, $2),10 $3), because sketches can be reused across different $1+$2 and $3 settings. *Space usage*: Fig. 16j shows the space required for storing the sketches and views created after evaluating all 200 queries. PS needs less than 0.3MB space whereas the MVs occupy between 2.6GB and 45GB. Even if storage is not a limiting factor, these views will compete with table data for available buffer pool space.

11.9 SQL Code for Queries used in the Experiments

11.9.1 Synthetic Data. SYN-Q1.

```
SELECT avg(a) as avga, ee
FROM v2
GROUP BY ee
HAVING avg(a) > $1;
```

For the MV experiments we materialize the subquery shown below.

```
SELECT avg(a) as avga, ee
FROM v2
GROUP BY ee
```

SYN-Q2.

```
SELECT maxa,v3.ee
FROM (
    SELECT max(a) as maxa, ee
    FROM v2
    GROUP BY ee) y,
    v3
WHERE y.ee = v3.ee AND maxa > $1
```

For the MV experiments we materialize the subquery shown below.

```
SELECT max(a) as maxa, ee
FROM v2
GROUP BY ee;
```

11.9.2 Crimes Data. C-Q1.

```
SELECT *
FROM (
  SELECT count(*) AS cnt, district, ward, block, community_area, beat
  FROM crimes
  GROUP BY district, ward, block, community_area, beat) f
ORDER BY cnt DESC
LIMIT 5;
```

C-Q2.

```
SELECT count(*)
FROM (
  SELECT * FROM
    (SELECT *
    FROM
      (SELECT count(*) AS cnt, block
      FROM crimes
      GROUP  BY block) f1
    ) f2
  WHERE cnt > 10000) f3;
```

11.9.3 Movies Data. M-Q1.

```
SELECT title, movieid, num_ratings
FROM (
  SELECT title, m.movieid , count(rating) AS num_ratings
  FROM movies m, ratings r
  WHERE m.movieid=r.movieid
  GROUP BY m.movieid, title) n
ORDER BY num_ratings DESC
LIMIT 10;
```

M-Q2.

```
SELECT COUNT(*) AS cnt
FROM (
  SELECT title, m.movieid , count(rating) AS num_ratings
  FROM movies m, ratings r
  WHERE m.movieid=r.movieid
  GROUP BY m.movieid, title) c1
WHERE num_ratings >=63300;
```

M-Q3.

```
SELECT title, m.movieid, total_cnt
FROM (
  SELECT total_cnt, movieid
  FROM
    (SELECT total_cnt, movieid
    FROM (
      SELECT cnt1+cnt2 AS total_cnt, t1.movieid
      FROM
        (SELECT count(*) AS cnt1, movieid
        FROM ratings
        GROUP BY movieid) r1,
        (SELECT count(*) AS cnt2, movieid
        FROM tags
        GROUP BY movieid) t1
      WHERE r1.movieid = t1.movieid) tc1
    ORDER BY total_cnt DESC) tc2
  LIMIT 10) tc3,
  movies m
WHERE tc3.movieid = m.movieid;
```

11.9.4 *Stack Overflow Data.* S-Q1.

```
SELECT * FROM
  (SELECT u_id, count(*) as numPosts
  FROM users, posts
  WHERE u_id = p_owneruserid
GROUP BY u_id) a
ORDER BY numPosts DESC
LIMIT 10;
```

S-Q2.

```
SELECT * FROM (
  SELECT u_id, count(p_favoritecount) AS numCmts
  FROM users, comments, posts
  WHERE u_id = c_userid AND c_postid = p_id
  GROUP BY u_id ) a
ORDER BY numFavors DESC
LIMIT 10;
```

S-Q3.

```
SELECT * FROM
    (SELECT count(*) AS cnt,u_id,u_displayname
    FROM comments, users
    WHERE c_userid=u_id
    GROUP BY u_id,u_displayname) a
ORDER BY cnt
LIMIT 10;
```

S-Q4.

```
SELECT * FROM
  (SELECT count(*) AS cnt,u_id,u_displayname
  FROM badges, users
  WHERE b_userid=u_id
  GROUP BY u_id,u_displayname) a
ORDER BY cnt
LIMIT 10;
```

S-Q5.

```
SELECT * FROM
    (SELECT count(*) AS cnt,u_id,u_displayname
    FROM comments, users
    WHERE c_userid=u_id
    GROUP BY u_id,u_displayname) a
WHERE cnt >= 47945 AND cnt <= 52973;
```

S-Q6.

```
SELECT * FROM (
  SELECT * FROM (
    SELECT * FROM (
      SELECT u_Id,
            Count(p_id) AS Answers,
            AVG(p_score) as avg_score
      FROM posts_bb1, users_b1
      WHERE u_id = p_owneruserid and p_posttypeid = 2 and
      p_communityowneddate is null
      GROUP BY u_id, u_displayname) a
    WHERE Answers < 10) b
  ORDER BY avg_score DESC) c
LIMIT 100;
```

S-Q7.

```
SELECT * FROM (
  SELECT * from (
    SELECT  t_TagName AS Tags,
          avg(u_upvotes) AS avg_ans_age,
          count(*) AS num_ans
    FROM (
      SELECT DISTINCT pa.p_OwnerUserId, pt.t_Id
      FROM users u1, posts pa, posts pq, PostTags pt
      WHERE pa.p_OwnerUserId = u1.u_Id AND pq.p_Id = pa.p_ParentId
          AND pt.p_id = pq.p_Id) ut,
      tags t, users u
      WHERE t.t_Id = ut.t_Id AND u.u_Id = ut.p_OwnerUserId
      GROUP BY t.t_Id, t.t_TagName) a
  WHERE num_ans > 1000) b
ORDER BY 2 DESC;
```

S-Q8.

```
SELECT * FROM (
  SELECT count(p.p_id) as rate, t.t_id as tagname
  FROM tags t, posttags_b1 pt, posts_bb1 p
  WHERE t.t_id = pt.t_id AND p.p_id = pt.p_id AND
        p.p_creationdate < '2010-06-01' AND
        p.p_creationdate > '2010-05-01'
  GROUP BY t.t_id) as rate,
  (SELECT * FROM (
    SELECT count(p.p_id) as num, t.t_id as tagname
    FROM tags t, posttags_b1 pt, posts_bb1 p
    WHERE t.t_id = pt.t_id AND p.p_id = pt.p_id
    GROUP BY t.t_id) a WHERE num > 800) as num
WHERE rate.tagname = num.tagname;
```

S-Q9.

```
SELECT * FROM (
    SELECT p.p_id as p_link, up, down from (
        SELECT  v_postid,
                sum(case when v_votetypeid = 2 then 1 else 0 end) as up,
                sum(case when v_votetypeid = 3 then 1 else 0 end) as down
        FROM Votes
        WHERE v_votetypeid in (2,3)
        GROUP BY v_postid) x,
    posts p
    WHERE v_postid = p.p_id and down > (up * 0.5)
            AND p.p_communityowneddate is null
    ORDER BY up desc
) y LIMIT 100;
```

S-Q10.

```
SELECT C.u_id as u_link, sum(D.downvotes) AS downvotes
FROM Users AS C, (
  SELECT A.p_id as postId, A.p_owneruserid AS userId, B.downvotes
  FROM Posts AS A,(
    SELECT v_postid, count(*) AS downvotes
    FROM Votes
    WHERE v_votetypeid = 3
    GROUP BY v_postid
  ) AS B
  WHERE A.p_id = B.v_postid
) as D
WHERE C.u_id = D.userId
GROUP BY C.u_id
ORDER BY downvotes desc
LIMIT 100;
```

S-Q11.

```
SELECT * FROM (
  SELECT s.userId AS u_link, u.u_displayname, s.answerCount,
         u.u_reputation, u.u_reputation / s.answerCount as peranswer avg
  FROM (
    SELECT u.u_id AS userId, count(p.p_id) AS answerCount
    FROM users u, posts p
    WHERE u.u_id=p.p_owneruserid AND p.p_posttypeid = 2 GROUP BY u.u_id) s,
    users u
  WHERE s.userId=u.u_id
  ORDER BY s.answerCount DESC) x
LIMIT 10;
```

11.9.5 End-to-End Query Templates. SFO-T1.

```
SELECT * FROM
  (SELECT u_id, count(p_favoritecount) AS cnt
  FROM users, comments, posts
  WHERE u_id = c_userid AND c_postid = p_id
  GROUP BY u_id ) a
WHERE cnt >= $1  AND cnt <= $2
```

SFO-T2.

```
SELECT * FROM
  (SELECT count(*) AS cnt,u_id,u_displayname
  FROM badges, users
  WHERE b_userid=u_id
  GROUP BY u_id,u_displayname) a
WHERE cnt >= $1 and cnt <= $2
```

SFO-T3.

```
SELECT * FROM
    (SELECT count(*) AS cnt,u_id,u_displayname
    FROM comments, users
    WHERE c_userid=u_id
    GROUP BY u_id,u_displayname) a
WHERE cnt >= $1 AND cnt <= $2
```

SFO-T1, SFO-T2 and SFO-T3 are used in the mixed end-to-end experiment for Fig. 13e. SFO-T1 is also used for end-to-end experiment by varying the query selectivities (0.7%, 2% and 5%) (Fig. 13f to 13h). SFO-T3 is also used for end-to-end experiment by varying the SDV (1k and 5k) (Fig. 13c and 13d).

C-T1.

```
SELECT *
FROM (
  SELECT primary_type, count(*) AS C
  FROM crimes
  GROUP BY primary_type)
WHERE C >= $1 AND C < $2
```

C-T2.

```
SELECT *
FROM (
  SELECT district,ward,cblock,community_area,beat,count(*) as cnt
  FROM crimes
  WHERE ward >= $1 AND ward < $2
  GROUP BY district,ward,cblock,community_area,beat) a
WHERE cnt >= $3 and cnt <= $4
```

C-T3.

```
SELECT *
FROM (
  SELECT *
  FROM (
    SELECT district,ward,cblock,community_area,beat, count(*) AS cnt
    FROM crimes
    WHERE cyear >= $1
    GROUP BY district,ward,cblock,community_area,beat) a
  ORDER BY cnt DESC) b
LIMIT $2
```

C-T4.

```
SELECT *
FROM (
  SELECT district,ward,cblock,community_area,beat, count(*) AS C
  FROM crimes
  WHERE cyear >= $1 AND cyear <= $2
  GROUP BY district,ward,cblock,community_area,beat) a
WHERE C >= $3
```

C-T5.

```
SELECT *
FROM (
  SELECT district,ward,cblock,community_area,beat,count(*) AS cnt
  FROM crimes
  WHERE community_area >= $1 AND community_area < $2
        AND cyear >= $3
  GROUP BY district,ward,cblock,community_area,beat) a
WHERE cnt >= $4 AND cnt <= $5
```

Query C-T1 is used in the end-to-end experiment by varying the query selectivities (5%, 10% and 20%) (Fig. 13b). Queries C-T2, C-T3, C-T4 and C-T5 are used in the mixed templates end-to-end experiment (Fig. 13a).

12 CONCLUSIONS AND FUTURE WORK

We present provenance-based data skipping (PBDS), a novel technique that determines at runtime which data is relevant for answering a query and then exploits this information to speed-up future queries. PBDS uses provenance sketches to concisely over-approximate the data that is relevant for a query. We develop self-tuning techniques for reusing a sketches captured for one query to answer a different query. PBDS results in significant performance improvements for important classes of queries such as top-k queries that are highly selective, but where it is not possible to determine statically what data is relevant. In the future, we will investigate how to maintain provenance sketches under updates and extend our self-tuning techniques to support wider range of queries and more powerful strategies.

REFERENCES

[1] [n.d.]. http://www.tpc.org/tpch/.

[2] Serge Abiteboul and Olivier Duschka. 2013. Complexity of Answering Queries Using Materialized Views. (2013).

[3] Serge Abiteboul and Oliver M Duschka. 1998. Complexity of answering queries using materialized views. In *PODS*. 254–263.

[4] Daniar Achakeev and Bernhard Seeger. 2013. Efficient bulk updates on multiversion B-trees. *PVLDB* 6, 14 (2013), 1834–1845.

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

[6] S. Agrawal, V. Narasayya, and B. Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*. 359–370.

[7] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. 2008. A practical scalable distributed B-tree. *PVLDB* 1, 1 (2008), 598–609.

[8] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB* 5, 10 (2012), 968–979.

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

[10] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. 2003. Scalable template-based query containment checking for web semantic caches. In *ICDE*. 493–504.

[11] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for Aggregate Queries. In *PODS*. 153–164.

[12] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram LudÃd'scher. 2009. Efficient Provenance Storage over Nested Data Collections. In *EDBT*. 958–969.

[13] Kamel Aouiche, JÃľrÃť me Darmont, Omar Boussaid, and Fadila Bentayeb. 2005. Automatic Selection of Bitmap Join Indexes in Data Warehouses. In *DaWaK*. 64–73.

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

[15] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Val Tannen. 2016. Algorithms for Provisioning Queries and Analytics. In *ICDT*. 18:1–18:18.

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

[17] C. BÃűhm, S. Berchtold, H.P. Kriegel, and U. Michel. 2000. Multidimensional index structures in relational databases. *Journal of Intelligent Information Systems* 15, 1 (2000), 51–70.

[18] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal data partitioning in database design. In *SIGMOD*. 128–136.

[19] Ashok K Chandra and Philip M Merlin. 1977. Optimal implementation of conjunctive queries in relational data bases. In *STOC*. 77–90.

[20] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. 2008. Efficient Provenance Storage. In *SIGMOD*. 993–1006.

[21] Surajit Chaudhuri, Mayur Datar, and Vivek Narasayya. 2004. Index selection for databases: A hardness study and a principled heuristic solution. *TKDE* 16, 11 (2004), 1313–1323.

[22] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing queries with materialized views. In *ICDE*. 190–190.

[23] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning database systems: a decade of progress. In *VLDB*. 3–14.

[24] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2017. Distributed Provenance Compression. In *SIGMOD*. 203–218.

[25] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.

[26] John Clarke. 2013. Storage indexes. In *Oracle Exadata Recipes*. 553–576.

[27] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *TODS* 25, 2 (2000), 179–227.

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[29] Leonardo MendonÃğa de Moura and Nikolaj BjÃÿrner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

[30] D. Deutch, Z. Ives, T. Milo, and V. Tannen. 2013. Caravan: Provisioning for What-If Analysis. *CIDR* (2013).

[31] Daniel Deutch, Yuval Moskovitch, and Noam Rinetzky. 2019. Hypothetical Reasoning via Provenance Abstraction. In *SIGMOD*. 537–554.

[32] Daniel Deutch, Yuval Moskovitch, and Val Tannen. 2013. PROPOLIS: Provisioned Analysis of Data-Centric Processes. *PVLDB* 6, 12 (2013).

[33] Jiang Du. 2013. DeepSea: self-adaptive data partitioning and replication in scalable distributed data systems. In *PODS*. 7–12.

[34] Jiang Du, Boris Glavic, Wei Tan, and RenÃľe J. Miller. 2017. DeepSea: Adaptive Workload-Aware Partitioning of Materialized Views in Scalable Data Analytics. In *EDBT*. 198–209.

[35] Kayhan Dursun, Carsten Binnig, Ugur ÃĞetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). 1275–1289.

[36] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and informative explanations of outcomes. *PVLDB* 8, 1 (2014).

[37] Kareem El Gebaly, Guoyao Feng, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2018. Explanation Tables. *Sat* 5 (2018), 14.

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

[39] F. Geerts and A. Poggi. 2010. On database query languages for K-relations. *Journal of Applied Logic* 8, 2 (2010), 173–185.

[40] Boris Glavic. 2021. Data Provenance - Origins, Applications, Algorithms, and Models. *Foundations and TrendsÂő in Databases* 9, 3-4 (2021), 209–441. https://doi.org/10.1561/1900000068

[41] Boris Glavic, Sven KÃűhler, Sean Riddle, and Bertram LudÃd'scher. 2015. Towards Constraint-based Explanations for Answers and Non-Answers. In *TaPP*.

[42] Boris Glavic, RenÃľe J Miller, and Gustavo Alonso. 2013. Using SQL for Efficient Generation and Querying of Provenance Information. In *In Search of Elegance in the Theory and Practice of Computation*. 291–320.

[43] J. Goldstein and P.ÃĚ. Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Record* 30, 2 (2001), 331–342.

[44] Goetz Graefe. 2006. B-tree indexes for high update rates. *SIGMOD Record* 35, 1 (2006), 39–44.

[45] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*. 371–381.

[46] Todd J Green, Molham Aref, and Grigoris Karvounarakis. 2012. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*. 1–8.

[47] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. 31–40.

[48] Todd J Green and Val Tannen. 2017. The Semiring Framework for Database Provenance. In *PODS*. 93–99.

[49] A. Gupta and I.S. Mumick. 1999. *Materialized views: techniques, implementations, and applications*. MIT press.

[50] Alon Y Halevy. 2001. Answering queries using views: A survey. *VLDB* 10, 4 (2001), 270–294.

[51] Thomas Heinis and Gustavo Alonso. 2008. Efficient Lineage Tracking for Scientific Workflows. In *SIGMOD*. 1007–1018.

[52] SÃąndor HÃľman, Niels J Nes, Marcin Åżukowski, and Peter Alexander Boncz. 2008. *Positional Delta Trees to reconcile updates with read-optimized data storage*. CWI. Information Systems [INS].

[53] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB* 4, 9 (2011), 586–597.

[54] Robert Ikeda, Semih Salihoglu, and Jennifer Widom. 2010. *Provenance-Based Refresh in Data-Oriented Workflows*. technical report.

[55] Robert Ikeda and Jennifer Widom. 2010. Panda: A System for Provenance and Data. In *TaPP '10*.

[56] Alekh Jindal and Jens Dittrich. 2012. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*. 65–80.

[57] G. Karvounarakis and T.J. Green. 2012. Semiring-Annotated Data: Queries and Provenance. *SIGMOD* 41, 3 (2012), 5–14.

[58] Anthony Klug. 1988. On conjunctive queries containing inequalities. *JACM* 35, 1 (1988), 146–160.

[59] S. KÃűhler, B. LudÃd'scher, and Y. Smaragdakis. 2012. Declarative datalog debugging for mere mortals. *Datalog in Academia and Industry* (2012), 111–122.

[60] Seokki Lee, Bertram LudÃd'scher, and Boris Glavic. 2018. Provenance Summaries for Answers and Non-Answers. *PVDLB* 11, 12 (2018), 1954–1957.

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

[62] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.

[63] Justin Levandoski, David Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. 2014. Indexing on modern hardware: Hekaton and beyond. In *SIGMOD*. 717–720.

[64] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[65] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. 1995. Answering queries using views. In *PODS*. 95–104.

[66] Xiang Li, Xiaoyang Xu, and Tanu Malik. 2016. Interactive provenance summaries for reproducible science. In *eScience*. 355–360.

[67] Zhe Li and Kenneth A. Ross. 1999. Fast Joins Using Join Indices. *VLDBJ* 8, 1 (1999), 1–24.

[68] T. Malik, L. Nistor, and A. Gehani. 2010. Tracking and Sketching Distributed Data Provenance. In *eScience*. 190–197.

[69] Guido Moerkotte. 1998. Small materialized aggregates: A light weight index structure for data warehousing. (1998).

[70] Tobias MÃijller, Benjamin Dietrich, and Torsten Grust. 2018. You Say 'What', I Hear 'Where'and 'Why'—(Mis-) Interpreting SQL to Derive Fine-Grained Provenance. *PVLDB* 11, 11 (2018).

[71] S.B. Navathe and M. Ra. 1989. Vertical partitioning for database design: a graphical algorithm. In *SIGMOD*. 450.

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

[73] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. 2018. Heuristic and Cost-based Optimization for Diverse Provenance Tasks. *TKDE* 31, 7 (2018), 1267–1280.

[74] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Record* 45, 2 (2016), 5–16.

[75] Dan Olteanu and Jakub ZÃąvodnÃ¡. 2011. On Factorisation of Provenance Polynomials. In *TaPP*.

[76] Patrick O'Neil and Goetz Graefe. 1995. Multi-table joins through bitmapped join indices. *SIGMOD Record* 24, 3 (1995), 8–11.

[77] S. Papadomanolakis and A. Ailamaki. 2004. Autopart: Automating schema design for large scientific databases using data partitioning. (2004).

[78] Luis L. Perez and Christopher M. Jermaine. 2014. History-aware Query Optimization with Materialized Intermediate Views. In *ICDE*.

[79] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained lineage at interactive speed. *PVLDB* 11, 6 (2018), 719–732.

[80] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*. 315–330.

[81] Sudeepa Roy, Laurel Orr, and Dan Suciu. 2015. Explaining query answers with explanation-ready databases. *PVLDB* 9, 4 (2015), 348–359.

[82] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries.

[83] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences among relational expressions with the union and difference operators. *JACM* 27, 4 (1980), 633–655.

[84] T. Sellis, N. Roussopoulos, and C. Faloutsos. 1987. The R-tree: A dynamic index for multi-dimensional objects. *VLDB* (1987), 507–518.

[85] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: provenance and probability management in postgreSQL. *PVLDB* 11, 12 (2018), 2034–2037.

[86] Lefteris Sidirourgos and Martin L. Kersten. 2013. Column imprints: a secondary index structure. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). 893–904.

[87] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *PVLDB* 10, 4 (2016), 421–432.

[88] Patrick Valduriez. 1987. Join Indices. *TODS* 12, 2 (1987), 218–246.

[89] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB* 6, 8 (2013), 553–564.

[90] Jia Yu and Mohamed Sarwat. 2016. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. *PVLDB* 10, 4 (2016), 385–396.

[91] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*. 1060–1071.

[92] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.

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