Provenance-based Data Skipping

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

ABSTRACT

Database systems use static analysis to determine upfront which data is needed for answering a query 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, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, Danica Porobic. Provenance-based Data Skipping. PVLDB, 15(3): 451 - 464, 2022.
doi:10.14778/3494124.3494130

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IITDBGroup/2021_pbds_reproducibility.

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 (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.

Consider a query Q_1: `SELECT city, popden FROM cities WHERE state = 'CA'` which returns the population density of cities in CA. The `WHERE` clause condition of this query implies that only rows fulfilling the condition `state = 'CA'` are **relevant**. The DBMS may use an index on column `state`, if it exists, to identify cities in California, to reduce 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.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494130

Q_2:
```
SELECT state, avg(popden) AS avgden
FROM cities
GROUP BY state
ORDER BY avgden DESC LIMIT 1;
```

$Q_2[\mathcal{P}_{state}]$:
```
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	
t_2	6000	San Diego	CA	f_1
t_3	5000	Sacramento	CA	
t_4	7000	New York	NY	
t_5	2000	Buffalo	NY	f_3
t_6	3700	Austin	TX	
t_7	2500	Houston	TX	f_4

(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_{state} : f_1 = [AL,DE], f_2 = [FL,MI], f_3 = [MN,OK], f_4 = [OR,WY]

F_{popden} : g_1 = [1000,4000], g_2 = [4001,9000]

(e) Range partitions of cities on `state` (top) and `popden` (bottom)

Figure 1: Running Example

EXAMPLE 1. *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.*

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 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. 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 2. *Using Lineage [25, 40], the provenance of query Q_2 from Ex. 1 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.

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 3. *Consider the sketch $\mathcal{P}_{state} = \{f_1\}$ from Ex. 2. 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 many fragments can result in conditions with many 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 4. *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]. *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.

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 what data is relevant 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 PBDS significantly improves performance. 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.

2 RELATED WORK

Physical Design and Self-tuning. Index structures [4, 7, 13, 17, 21, 44, 45, 51, 52, 61–63, 84, 84, 90], horizontal and vertical partitioning techniques [6, 18, 33, 55, 70, 77, 80, 87, 91], zone maps [26, 68], materialized views [3, 5, 8, 22, 43, 49, 50, 64], join indexes [13, 66, 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, 64, 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. 9.5, 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.

Provenance Capture. Many approaches for provenance capture, encode provenance as annotations on data and propagate such annotations through queries [46, 56, 71, 72]. Systems that capture database provenance include Perm [42], GProM [14], DBNotes [16], LogicBlox [46], Smoke [79], declarative Datalog debugging [58], ExSPAN [93], ProvSQL [85], Müller et al.'s approach [69], 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, 67, 74, 75] avoids storing common substructures in the provenance more than once. Closely related are techniques for provenance summarization [9, 31, 41, 59, 60, 65], 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, 60, 82]. Such summaries are typically not sufficient for our purpose, i.e., they may not encode a superset of the provenance or can not be effectively encoded as selection conditions.

Optimizing Operations with Provenance. Pandas [53, 54] uses provenance to selectively update the outputs 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 lightweight index that allows us to efficiently access relevant inputs.

3 BACKGROUND AND NOTATION

In this section we introduce provenance sketches and necessary background on provenance and range-partitioning.

Relational Algebra. We use bold face (non-bold) to denote relation and database schemas (instances). 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 of 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

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

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

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

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

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

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

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

Figure 2: Bag Relational Algebra

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 based 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 and breaking ties using the remaining attributes. The top-k operator $\tau_{O,C}(R)$ returns the C smallest tuples from R wrt. $<_O$.

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.

3.1 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.

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 and $\mathcal{R} = \{r_1, \ldots, r_n\}$ 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:*

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

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 $\mathcal{P} \supseteq \mathcal{R}(D, F_{\mathcal{R},a}(R), Q)$. A sketch is called* ***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, $\mathcal{P} = \{f_1\}$.

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. 4 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 and D a database. We call a set of sketches $\mathcal{PS}$* safe *for Q and D iff $Q(D_{\mathcal{PS}}) = Q(D)$.*

Obviously, only safe sketches are of interest. In Sec. 6 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 all 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, every sketch $\mathcal{P}$ based on $F_{\mathcal{R},A}$ is 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.*

4 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 the operators of a query such that each (intermediate) query result is annotated with its provenance.

EXAMPLE 5. *To capture the Lineage of each result tuple of the query Q_2 from Ex. 2 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`.

4.1 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 $\text{INIT}_{F_i}(R_i)$ 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 $\text{SNG}(i)$ 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:

$$\text{INIT}_F(R) := \Pi_{\mathbf{R}, \mathsf{select}(a \in r_1 \mapsto \text{SNG}(1), \ldots, a \in r_n \mapsto \text{SNG}(n))}(R) \quad (1)$$

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}.

$$\text{PROP}(\mathcal{F}, R) = \text{INIT}_{\mathcal{F}}(R) \quad (r_0) \qquad \text{PROP}(\mathcal{F}, Q_1 \times Q_2) = \text{PROP}(\mathcal{F}, Q_1) \times \text{PROP}(\mathcal{F}, Q_2) \quad (r_4)$$

$$\text{PROP}(\mathcal{F}, \Pi_A(Q)) = \Pi_{A,\Lambda}(\text{PROP}(\mathcal{F}, Q)) \quad (r_1) \qquad \text{PROP}(\mathcal{F}, \tau_{O,C}(Q)) = \tau_{O,C}(\text{PROP}(\mathcal{F}, Q)) \quad (r_5)$$

$$\text{PROP}(\mathcal{F}, \sigma_\theta(Q)) = \sigma_\theta(\text{PROP}(\mathcal{F}, Q)) \quad (r_2) \qquad \text{PROP}(\mathcal{F}, Q_1 \cup Q_2) = \text{PROP}(\mathcal{F}, Q_1) \cup \text{PROP}(\mathcal{F}, Q_2) \quad (r_6)$$

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

$$\text{INSTR}(\mathcal{F}, Q) = \gamma_{bitor(\Lambda)\to\Lambda}(\text{PROP}(\mathcal{F}, Q)) \quad (r_7)$$

Figure 3: Instrumentation rules for sketch capture

```
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
```

Q_{INIT}

4.2 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 $\text{INSTR}(\mathcal{F}, Q)$ 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 $A \to B$ and function application, e.g., $f(A)$ denotes $f(a_1), \ldots, f(a_n)$. We assume the existence of an aggregation function *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 $\text{INSTR}(\cdot)$ are shown in Fig. 3. As the last step of the rewritten query $\text{INSTR}(\cdot)$ we apply *bitor* aggregation to merge the sketch annotations of the results of the query (r_7). The input to this aggregation is generated using $\text{PROP}(\mathcal{F}, Q)$ which recursively replaces operators in Q with an instrumented version.

Rule r_0 initializes the sketch annotations for relation R using $\text{INIT}(\cdot)$ as introduced in Sec. 4.1. 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. 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 $\text{INSTR}(\mathcal{F}, Q)(D)$ produces a safe sketch.*

The capture query for our running example query Q_2 is shown below. This query returns $\{1000\}$ which encodes the sketch $\{f1\}$. Recall that subquery Q_{INIT} was shown already in Sec. 4.1.

```
SELECT bitor(λ_F_state) AS λ_F_state
FROM (SELECT state,avg(popden) AS avgden, bitor(λ_F_state) AS λ_F_state
      FROM Q_INIT GROUP BY state ORDER BY avgden DESC LIMIT 1)
```

4.3 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 set bit 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. We call this the *delay* method. Furthermore, in Postgres, the *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 (the *No-copy* method). For MonetDB we implement *bitor* as a user-defined aggregation function in C.

4.4 Attribute and Partition Selection

Selecting Attributes. The choice of attributes A on which we are creating a sketch can significantly affect the sketch instance size. The most important factors are (i) does A have sufficiently many distinct values to support fine-grained sketches, (ii) can we exploit physical design to skip data for a sketch build on A, and (iii) how predictive are a tuple's A-values of the tuple belonging to the 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 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.

5 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 encoding the sketches as 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 $R[\mathcal{P}] := \sigma_{\bigvee_{j=1}^{m} a \in r_{i_j}}(R)$. For example, the query Q_2 in the running example would be rewritten into $Q_2[\mathcal{P}_{state}]$ (see Fig. 1a).

5.1 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 $state \in [Al, DE] \vee state \in [FL, MI]$. 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.

6 TESTING SKETCH SAFETY

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 X are safe before paying the cost of creating such sketches, we design a method which only accesses Q and basic statistics of D, specifically the minimum and maximum values of each column. Given X, Q and the statistics 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. Our algorithm is only sound, but not complete, because, as we show in [73], any sound and complete algorithm for this problem has to have full access to the database. As a convention we implicitly assume that variables are universally quantified unless explicitly stated otherwise.

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 $D \subseteq D' \Rightarrow Q(D) \subseteq Q(D')$). See [73] for the formal definition of monotonicity and the proof. 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. 4. 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.

6.1 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 6. *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 (which we denote 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 than the full group for NY. 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, the top-1 operator will filter out these groups.*

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 6 (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 $R \lesssim_\Psi R'$ 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:*

$$\forall t \in R : \exists t' \in R' : \mathcal{M}(t, t') \quad (1) \qquad \forall (t, t') \in \mathcal{M} : (t, t') \models \Psi \quad (2)$$

$$\forall t_1, t_2, t_1', t_2' : \mathcal{M}(t_1, t_1') \wedge \mathcal{M}(t_2, t_2') \wedge (t_1 = t_2 \vee t_1' = t_2') \rightarrow t_1 = t_2 \wedge t_1' = t_2' \quad (3)$$

Conditions (1) and (3) ensure that every tuple from R is "matched" to exactly one tuple from R'. Condition (2) 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 provenance sketch instances. 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. 6. The relationship between the results of the subquery Q_{agg} over D and $D_{\mathcal{P}}$ can be encoded as the generalized containment relationship $Q_{agg}(D_{\mathcal{P}}) \lesssim_{sd^{\mathcal{PS}} \leq sd \wedge state^{\mathcal{PS}} = state} Q_{agg}(D)$.

6.2 Inference Rules

Given a query Q and a set of attributes X from the database D, we construct a logical formula $gc(Q, X)$ which evaluates to true iff $Q(D_{\mathcal{PS}})$ is generalized contained in $Q(D)$ according to a formula $\Psi_{Q,X}$ for any set of sketches $\mathcal{PS}$ created on X for D. For instance, for an aggregation, $\Psi_{Q,X}$ encodes how the aggregation function results for D and $D_{\mathcal{PS}}$ are related to each other. Intuitively, $gc(Q, X)$ 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 $a < 10$. We demonstrate in [73] that this type of generalized containment (based on $gc(Q, X)$) does imply $Q(D_{\mathcal{PS}}) = Q(D)$. In the construction of $gc(Q, X)$ we make use of several auxiliary constructs:

pred(Q). We use $pred$ to record conditions which are fulfilled by all tuples produced by query Q and its subqueries. $pred$ is computed bottom-up. For instance, selection and join conditions are added to $pred$, since all tuples produced by such operators have to fulfill these conditions. For example, given $Q := \sigma_{a=5}(\Pi_a(\sigma_{b<4}(R)))$, then $pred(Q) = (a = 5 \wedge b < 4)$. Note that in $pred(R)$ we use database statistics to bound the values of tuples from input relation R. $min(a)$ ($max(a)$) denotes the smallest (largest) value in attribute a.

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

For simplicity, we assume that attribute names are unique. Furthermore, we use $conds(Q)$ to denote $pred(Q) \wedge expr(Q)$ and $Q^{\mathcal{PS}}$ to denote query Q applied to the instance of $\mathcal{PS}$. We define $gc(Q, X)$ 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 $gc(Q, X)$ holds is based on the root operator of Q and whether gc holds for the root's children. Because of space limitations, we only show some gc rules used in our examples in Fig. 4. The remaining $pred$, $expr$, and gc rules are shown in [73].

Query Q	$gc(Q, X)$
R	**true**
$\gamma_{f(a) \rightarrow b; G}(Q_1)$	$gc(Q_1, X_1) \wedge (\forall g \in G : \Psi_{Q_1, X_1} \wedge conds(Q_1^{\mathcal{PS}}) \wedge conds(Q_1) \rightarrow g^{\mathcal{PS}} = g)$
$\tau_{O,C}(Q_1)$	$gc(Q_1, X_1) \wedge (\forall o \in O : \Psi_{Q_1, X_1} \wedge conds(Q_1^{\mathcal{PS}}) \wedge conds(Q_1) \rightarrow o \leq o^{\mathcal{PS}})$

Figure 4: Exemplary rules for $gc(Q, X)$

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

Aggregation. In Fig. 4, 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 for Q_1 and the group-by attributes are equal for all inputs, then generalized containment will hold for the result. To determine $\Psi_{Q,X}$ 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:

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

(i) if X_1 is a set of attributes that is guaranteed to be equal to a subset of the group-by attributes, then calculating the aggregation function over the sketch instance yields the same result as over the 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, $\Psi_{Q,X}$ is the same as Ψ_{Q_1,X_1}.

EXAMPLE 7. *Reconsider query Q_{total} written in relational algebra: $\tau_{desc,1}(\Pi_{state,sd \cdot -1 \rightarrow desc}(\gamma_{state;sum(popden) \rightarrow sd}(cities)))$. Since the top-k operator uses ascending order, we have to encode* `DESC` *by multiplying sd with -1. To determine whether* `popden` *is a safe attribute for Q_{total} , we calculate $gc(Q_{total}, \{popden\})$ using the rules from Fig. 4. For relation* CITIES*, since $popden > 0$, then $pred(cities) = popden > 0$, $expr(cities) = \emptyset$, $\Psi_{cities,\{popden\}} =$*

$popden^{\mathcal{PS}} = popden \wedge city^{\mathcal{PS}} = city \wedge state^{\mathcal{PS}} = state$, *and* $gc(cities, \{popden\})$ *evaluates to true. Next,* $gc(Q_{agg}, \{popden\})$ *evaluates to true, because* $\Psi_{cities,\{popden\}}$ *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* $\Psi_{Q_{agg},\{popden\}}$. *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* $gc(Q_{total}, \{popden\})$ *evaluates to true. Hence, any sketch build on attribute popden is safe for this query.*

We now show that our safety check condition is correct, i.e., if $gc(Q, X)$ holds, then X is a safe set of attributes for Q.

THEOREM 2 ($gc(Q, X)$ IMPLIES SAFETY OF X). *Let Q be a query, D be a database, 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 $gc(Q, X)$ holds, then X is a safe for Q wrt. D.*

PROOF SKETCH. The claim is proven by first proving two lemmas that state that (i) $gc(Q, X)$ implies $gc(Q', X')$ for any subquery of Q (this follows trivially from the definition of gc) and that (ii) $gc(Q', X')$ implies $Q'(D_{\mathcal{PS}}) \lesssim_{\Psi} Q'(D)$ for any subquery Q' of Q. (ii) is proven by induction over the structure of a query. Then based on these results we prove the theorem by demonstrating that $gc(Q, X)$ together with the fact that $D_{\mathcal{PS}}$ contains the provenance of Q implies the claim. □

7 REUSING PROVENANCE SKETCHES

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, 57, 83]. We develop a solution for a 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. 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. 6.

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 result of substituting each 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 condition, we first state three observations. (i) The same sets of attributes are safe for all instances of a parameterized query. (ii) Adding additional fragments to a safe sketch $\mathcal{P}$ for a query Q yields a safe sketch (Sec. 6). (iii) 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 $P(Q, D)$ ($P(Q', D)$), two sets of accurate provenance sketches $\mathcal{PS}$ and $\mathcal{PS}'$ build over the same attributes X and partitions such that $\mathcal{PS}$ ($\mathcal{PS}'$) is a sketch for Q (Q'). If $P(Q, D) \supseteq P(Q', D)$ then $\mathcal{PS} \supseteq \mathcal{PS}'$ and, thus, also $D_{\mathcal{PS}} \supseteq D_{\mathcal{PS}'}$.

Based on these observations, we prove three lemmas that 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*. We refer the interested reader to [73] for the details. 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. 6. Our condition consists of two parts: $uconds(Q', Q)$ (shown below) and $ge(Q', Q)$. Condition ge (Fig. 5) 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 ($Q'(D) \lesssim_{\Psi_{Q',Q}} Q(D)$). 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. 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 $Q = \sigma_{a=40}(\sigma_{a>30}(R))$ and $Q' = \sigma_{a=40}(\sigma_{a>10}(R))$. Subquery $\sigma_{a>10}(R)$ is not contained in $\sigma_{a>30}(R)$, but Q and Q' are equivalent. To be able to determine generalized containment, even if 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 $a = a' \wedge a' = 40 \wedge a' > 10 \rightarrow a = 40 \wedge a > 30$ 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 $non\text{-}grp\text{-}pred(Q)$ 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 a group-by attribute, we get $non\text{-}grp\text{-}pred(Q) = a > 10$. 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,

$$\Psi_{R',R} = \bigwedge_{a \in \text{SCH}(R)} a = a'$$

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

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

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

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

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

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

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

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

Figure 5: Rules defining $ge(Q', Q)$ and $\Psi_{Q',Q}$ which are used to test reusability

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.

EXAMPLE 8. *Consider the parameterized query* $\mathcal{T} = \sigma_{cnt>\$2}(\gamma_{state;count(*)\rightarrow cnt}(\sigma_{popden>\$1}(cities)))$. *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 answer* Q', *we construct the conditions shown below. We use* p, c, *and* s *to denote* $popden$, $city$, *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 tests 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'.

We now demonstrate that our approach is sound.

THEOREM 3. *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 SKETCH. We first demonstrate that $ge(Q', Q) \wedge uconds(Q', Q)$ implies that the generalized containment $Q'(D) \preceq_{\Psi_{Q',Q}} Q(D)$ holds, thus, establishing a connection between all tuples in $Q'(D)$ and tuples of $Q(D)$. We then show that given an arbitrary mapping $\mathcal{M}$ based on which $Q'(D) \preceq_{\Psi_{Q',Q}} Q(D)$, for any $(t', t) \in \mathcal{M}$, the provenance of $t' \in Q'(D)$ is a subset of the provenance of $t \in Q(D)$. By definition of generalized containment, for all $t' \in Q'(D)$ there has to exist $t \in Q(D)$ such that $(t', t) \in \mathcal{M}$ which immediately implies that $P(Q', D) \subseteq P(Q, D)$. Since we have shown before that provenance containment implies that $\mathcal{PS}$ is safe for Q', this concludes the proof. For the detailed proof, please see [73]. □

8 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. Our strategy decides 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. 7 to determine whether one of the sketches we have captured can be used to answer an incoming query.

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*. In [73] we also evaluate an *eager strategy* that creates new sketches whenever a query cannot use any of the existing sketches. We keep track of sketches we have captured by mapping pairs of parameterized queries and parameter bindings to the sketches we have created for these queries and parameter bindings.

9 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 preliminary experiments, we have evaluated the optimizations for sketch capture from Sec. 4.3, demonstrating that using binary search to determine which partition a tuple belongs to and representing singleton sketches

as integers instead of sets is always beneficial (see [73]). Thus, we always applied these optimizations. In all experiments, we determined what attributes are safe using the techniques from Sec. 6. For experiments measuring the end-to-end performance of PBDS, the cost of safety and reuse checks is included in the runtime.

9.1 Workloads and Datasets

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

Stack Overflow. 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. The remaining queries and SQL code for all queries are shown in [73].

9.2 TPC-H

Because of the TPC-H benchmark's artificial data distribution, this stresses our approach since there are essentially no meaningful correlations that we can exploit. As explained in Sec. 4.4 , 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 selectivity of sketches (the fraction of input data covered by the sketch). We show these sketch selectivities in [73]. For many queries we already achieve selectivities of a few percent for PS4000 (4000 fragments). 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.

Postgres - Capture & Reuse. Fig. 6a and 6f 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 fragment of the partition (Sec. 5).

Fig. 6a 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 customers. 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. For Q5, Q7, Q8, Q20 and Q21 more fine-grained partitioning is required to benefit from PBDS. While Q2 and Q17 have selective sketches, their selection conditions are quite restrictive leaving little room for improvement. Fig. 6f shows runtimes for SF10. Observe that the runtime of queries Q2, Q3, Q10, Q20 and Q21 exhibit similar behavior as for SF1.

Fig. 6b and 6g 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. In [73] we analyze after how many executions of a query the cost of capturing a sketch has been amortized (for most queries after using the sketch once or twice). Overall, partitions with 10,000 fragments provide the best trade-off between capture and use performance for our workloads.

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. 6h and 6i show the runtime for using sketches. Even though MonetDB supports database cracking [52] and column imprints [86] (a technique similar to zone maps), the implementation of these techniques turned out to not be beneficial for PBDS (see [73] for a detailed explanation). 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. 6j and 6k shows the relative overhead of sketch capture (similar trends as for Postgres). We omit PS100000 since it did not result in additional improvement.

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. 6c. 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 *DB-X* cannot utilize binary search when using a zonemap to skip data. This is also why Q19 and Q20 runtimes are slower than No-PS that the provenance sketches for these queries are quite large. 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 (we omit these results).

9.3 Stack Overflow Dataset

Since this is a large dataset, we only consider 1000 and 10000 fragments. Fig. 6d shows that PBDS improves query performance by 96.9% to 98.85% for PS10000. The capture overhead ranges between a factor of ~ -0.14 and ~ 1.2 (Fig. 6e). The negative overhead is caused by Postgres choosing parallel pre-aggregation for the capture query, but not for the No-PS query. We show results for additional real world datasets in [73].

9.4 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

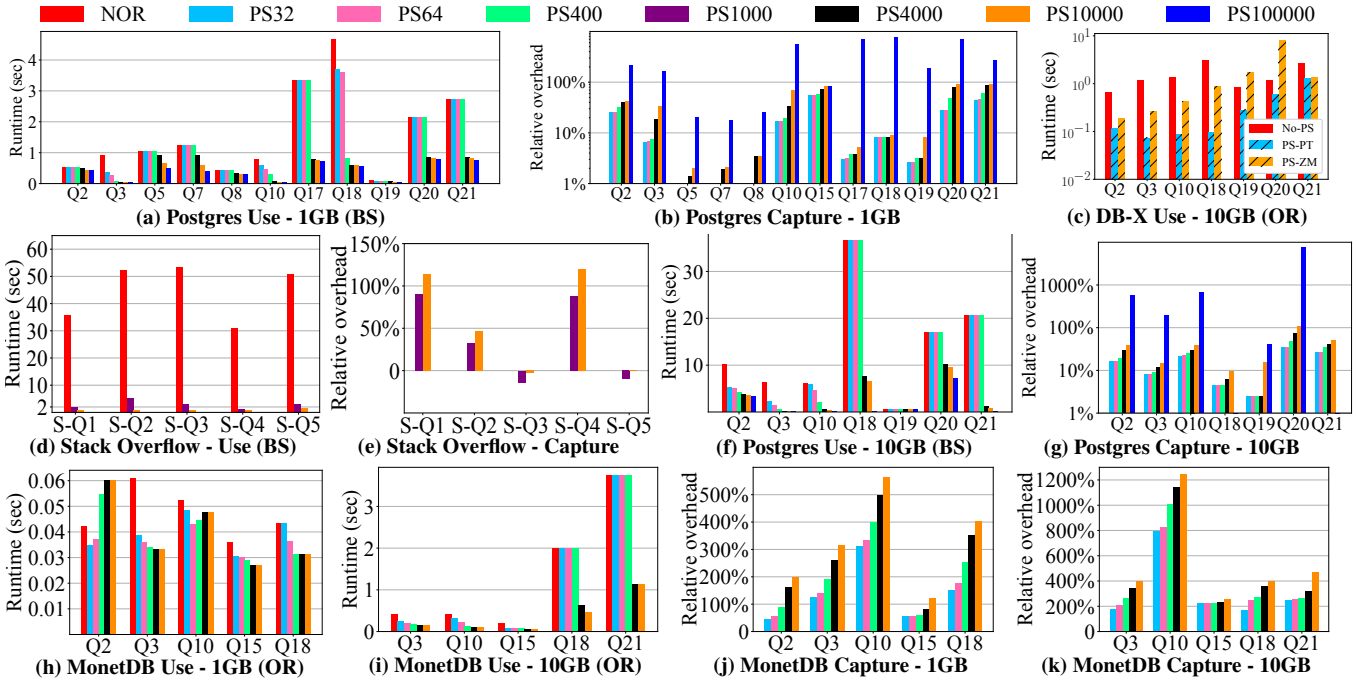


Figure 6: Performance of provenance sketch capture and use for TPC-H and Stack Overflow queries.

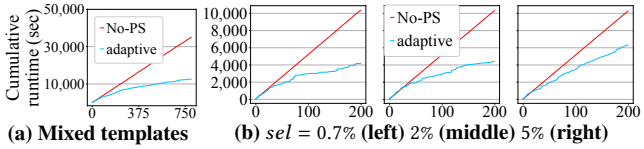


Figure 7: End-to-end experiments on stack overflow data. We report the cumulative runtime in sec (the x-axis show the number of queries that have been executed up to that point).

using the techniques described in Sec. 8. Note that the runtimes reported here include the runtime of safety and reuse checking and the cost of creating sketches. Given a set of query templates, we generate workloads by randomly choosing for each query the template and parameter values controlling for average query selectivity (*sel*). For this experiment we modified queries introduced in Sec. 9.1 by changing `LIMIT` to a `HAVING`.

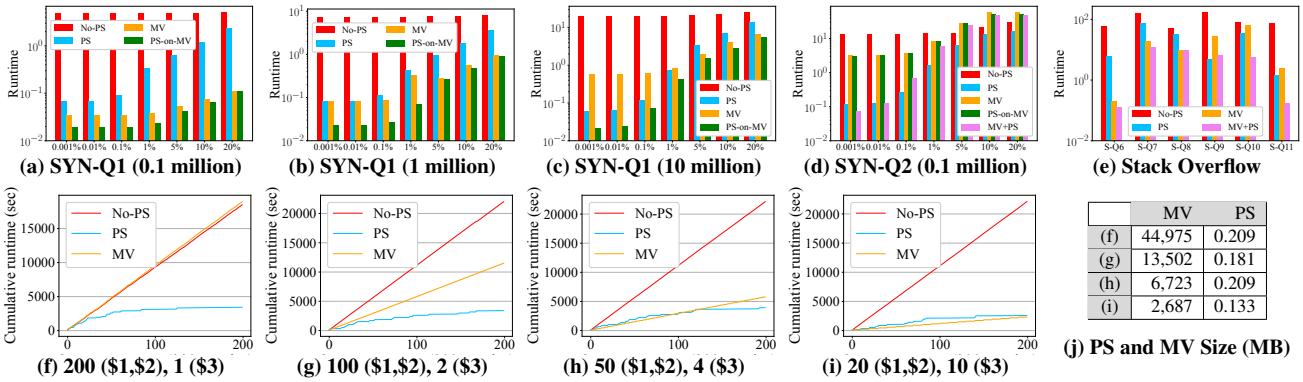
Stack Overflow. Fig. 7a shows the result of a workload over the stack overflow dataset with three query templates (SQL code shown in [73]). We set $sel = 1\%$. Since our *adaptive* strategy (see Sec. 8) delays capturing sketches and we pay for creating sketches, there is a delay before we see benefits. 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. 7b 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 vary the standard deviation (*SDV*) of the normal distribution and show the result in [73].

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.

9.5 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.

Synthetic Datasets. We start with a synthetic dataset with 40M rows. 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. 8a to 8c show the runtime for answering SYN-Q1 for MVS from 0.1 (Fig. 8a) to 10 million (Fig. 8c) 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



(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 8: 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).

million (fig. 8a), MV outperforms PS since we only need to evaluate the having condition on the small MV. For MVS 10 million, PS is better when selectivity < 5%. Building a sketch on the MV (PS-on-MV) outperforms both methods for this template. Fig. 8d 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. 8d), 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.

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. 8e. Note that 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. Running the top-k operation on the materialized aggregation (MV) performs best. 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 applies 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 that 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 MV for these queries, because PSSR is relatively low (top-k query) and we can use sketches to filter joined tables. Notably, the hybrid approach (PS+MV) outperforms both PS and MV for most queries.

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. 8) 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. *Results*: Results are shown in Fig. 8. Both methods are highly efficient if an existing sketch/view can be reused. Thus, the main cost 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. 8j 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.

10 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.

Acknowledgments. This work is supported in part by NSF awards IIS-1956123 and IIS-2107107.

REFERENCES

[1] [n.d.]. http://www.tpc.org/tpch/ (visited on 10/28/2021).

[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 Ludäscher. 2009. Efficient Provenance Storage over Nested Data Collections. In *EDBT*. 958–969.

[13] Kamel Aouiche, Jérôme 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. Böhm, 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 Mendonça de Moura and Nikolaj Bjørner. 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 Renée 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 Çetintemel, 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 Trends® in Databases* 9, 3-4 (2021), 209–441. https://doi.org/10.1561/1900000068

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

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

[43] J. Goldstein and P.Å. 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] Sándor Héman, Niels J Nes, Marcin Żukowski, and Peter Alexander Boncz. 2008. *Positional Delta Trees to reconcile updates with read-optimized data storage*. CWI. Information Systems [INS].

[52] 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.

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

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

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

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

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

[58] S. Köhler, B. Ludäscher, and Y. Smaragdakis. 2012. Declarative datalog debugging for mere mortals. *Datalog in Academia and Industry* (2012), 111–122.

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

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

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

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

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

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

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

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

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

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

[69] Tobias Müller, 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).

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

[71] 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.

[72] 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.

[73] Xing Niu, Ziyu Liu, Pengyuan Li, and Boris Glavic. 2021. Provenance-based Data Skipping (extended version). (2021). arXiv:2104.12815

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

[75] Dan Olteanu and Jakub Závodný. 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.