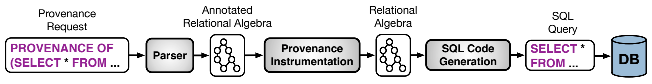
Optimizing Provenance Computations

Xing Niu and Boris Glavic

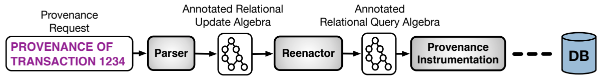
IIT DB Group Technical Report
IIT/CS-DB-2016-02

2016-10

http://www.cs.iit.edu/~dbgroup/

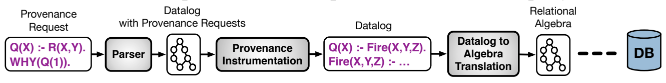


(a) Provenance is captured using an annotated semantics of relational algebra which is compiled into standard relational algebra over a relational encoding of annotated relations and then translated into SQL code.



Provenance Request
PROVENANCE OF TRANSACTION 1234
Parser
Annotated Relational Update Algebra
Reenactor
Annotated Relational Query Algebra
Provenance Instrumentation
DB

(b) In addition to the steps of **(a)**, this pipeline contains a step called *reenactment* that compiles annotated updates into annotated query semantics.



Provenance Request
Q(X) :- R(X,Y). WHY(Q(1)).
Parser
Datalog with Provenance Requests
Provenance Instrumentation
Datalog
Q(X) :- Fire(X,Y,Z). Fire(X,Y,Z) :- ...
Datalog to Algebra Translation
Relational Algebra
DB

(c) Computing the edge relation of provenance graphs for Datalog queries based on a rewriting called *firing rules*. The instrumented Datalog program is compiled into relational algebra which in turn is translated into SQL.

Fig. 1: Instrumentation pipelines for capturing provenance for **(a)** SQL queries, **(b)** transactions, and **(c)** Datalog queries.

	shop	item
a_1	Walmart	Steak
a_2	Walmart	Butter
a_3	Walmart	Bread
a_4	Cosco	Butter
a_5	Cosco	Bread

(a) sale

	name	numEmpl
s_1	Walmart	3
s_2	Cosco	14

(b) shop

	id	price
i_1	Steak	100
i_2	Butter	10
i_3	Bread	25

(c) item

	name
$s_1 \cdot a_1 \cdot i_1$ $+s_1 \cdot a_3 \cdot i_3$	Walmart
$s_2 \cdot a_5 \cdot i_3$	Cosco

(d) Annotated result of Q

	result	**prov. shop**		**prov. sales**		**prov. items**	
	name	P(name)	P(numEmpl)	P(shop)	P(item)	P(id)	P(price)
$s_1 \cdot a_1 \cdot i_1$	Walmart	Walmart	3	Walmart	Steak	Steak	100
$s_1 \cdot a_3 \cdot i_3$	Walmart	Walmart	3	Walmart	Bread	Bread	25
$s_2 \cdot a_5 \cdot i_3$	Cosco	Cosco	14	Cosco	Bread	Bread	25

(e) Relational encoding of query Q's provenance

Fig. 2: Example database and query provenance

***Abstract*—Data provenance is essential for debugging query results, auditing data in cloud environments, and explaining outputs of Big Data analytics. A well-established technique is to represent provenance as annotations on data and to *instrument* queries to propagate these annotations to produce results annotated with provenance. However, even sophisticated optimizers are often incapable of producing efficient execution plans for instrumented queries, because of their inherent complexity and unusual structure. Thus, while instrumentation enables provenance support for databases without requiring any modification to the DBMS, the performance of this approach is far from optimal. In this work, we develop provenance-specific optimizations to address this problem. Specifically, we introduce algebraic equivalences targeted at instrumented queries and discuss alternative, equivalent ways of instrumenting a query for provenance capture. Furthermore, we present an extensible heuristic and cost-based optimization (CBO) framework that governs the application of these optimizations and implement this framework in our *GProM* provenance system. Our CBO is agnostic to the plan space shape, uses a DBMS for cost estimation, and enables retrofitting of optimization choices into existing code by adding a few LOC. Our experiments confirm that these optimizations are highly effective, often improving performance by several orders of magnitude for diverse provenance tasks.**

I. Introduction

Database provenance, information about the origin of data and the queries and/or updates that produced it, is critical for debugging queries, auditing, establishing trust in data, and many other use cases. The de facto standard for database provenance [16], [17], [14] is to model provenance as annotations on data and define an annotated semantics for queries that determines how annotations propagate. Under such a semantics, each output tuple t of a query Q is annotated with its provenance, i.e., a combination of input tuple annotations that explains how these inputs were used by Q to derive t.

Database provenance systems such as Perm [12], GProM [7], DBNotes [8], LogicBlox [14], declarative Datalog debugging [18], ExSPAN [25], and many others use a relational encoding of provenance annotations. These systems typically compile queries with annotated semantics into relational queries that produce this encoding of provenance annotations following the process outlined in Fig. 1a. We refer to this reduction from annotated to standard relational semantics as *provenance instrumentation* or instrumentation for short. The technique of compiling non-relational languages into relational languages (e.g., SQL) has also been applied for translating XQuery into SQL over a shredded representation of XML [15] and for compiling languages over nested collections into SQL [11]. The example below introduces a relational encoding of provenance polynomial [16] and the instrumentation approach for this model implemented in Perm [12].

Example 1. *Consider a query over the database in Fig. 2 returning shops that sell items which cost more than \$20:*

$$\Pi_{name}(shop \bowtie_{name=shop} sale \bowtie_{item=id} \sigma_{price>20}(item))$$

The result of this query is shown in Fig. 2d. Using provenance polynomials to represent provenance, tuples in the database are annotated with variables representing tuple identifier. We show these annotations to the left of each tuple. Each query result is annotated with a polynomial over these variables that explains how the tuple was derived by combining input tuples. The addition operation in these polynomials corresponds to alternative use of tuples such as in a union or projection and multiplication represents conjunctive use (e.g., a join). For example, the query result $(Walmart)$ *was derived by joining tuples* s_1*,* a_1*, and* i_1 $(s_1 \cdot a_1 \cdot i_1)$ *or alternatively by joining tuples* s_1*,* a_3*, and* i_3 $(s_1 \cdot a_3 \cdot i_3)$*. Fig. 2e shows a relational encoding of these annotations as supported by the Perm [12] and GProM [7] systems: variables are represented by the tuple they are annotating, multiplication is represented by concatenating the encoding of the factors, and addition is represented by encoding each summand as a separate tuple.*[1] *This encoding is computed by compiling the input query with annotated semantics into relational algebra. The resulting* instrumented *query is shown below. This query adds attributes from the input relations to the final projection and renames them (represented as* $\rightarrow$*) to denote that they store provenance.*

$$Q_{join} = shop \bowtie_{name=shop} sale \bowtie_{item=id} \sigma_{price>20}(item)$$
$$Q = \Pi_{name,name \rightarrow P(name),numEmp \rightarrow P(numEmp),\ldots}(Q_{join})$$

The instrumentation we are using here is defined for any SPJ (Select-Project-Join) query (and beyond) based on a set of algebraic rewrite rules (see [12] for details).

[1]The details are beyond the scope of this paper, e.g., the input polynomial is normalized into a sum of products. The interested reader is referred to [12].

A. Instrumentation Pipelines

Provenance for SQL Queries. The instrumentation technique shown in Fig. 1a and explained in the example above is applied by many relational provenance systems. For instance, the DBNotes [8] system uses instrumentation to propagate attribute-level annotations according to Where-provenance [10]. Variants of this particular instrumentation pipeline targeted in this work are discussed below. While the query used for provenance computation in Ex. 1 is rather straightforward and is likely to be optimized in a similar fashion as the input query, this is not true for more complex provenance computations.

Provenance for Transactions. Fig. 1b shows a pipeline used to retroactively capture the provenance of updates and transactions [5], [6] in GProM. In addition to the steps from Fig. 1a, this pipeline uses an additional compilation step called *reenactment*. Reenactment translates transactional histories with annotated semantics into equivalent temporal queries with annotated semantics. Such queries can be executed using any DBMS with support for time travel to capture the provenance of a past transaction. While the details of this approach are beyond the scope of this work, consider the following simplified SQL example. The SQL update `UPDATE R SET b = b + 2 WHERE a = 1` over relation `R(a,b)` can be reenacted using a query `SELECT a, CASE WHEN a=1 THEN b+2 ELSE b END AS b FROM R`. If executed over the version of the database seen by the update, this query is guaranteed to return the same result and have the provenance as the update.

Provenance for Datalog. The pipeline shown in Fig. 1c generates provenance graphs that explain why or why-not a tuple is in the result of a Datalog query [19]. Such graphs store which successful and failed rule derivations of the query were relevant for (not) deriving the (missing) result tuple of interest. This pipeline compiles such provenance requests into a Datalog program that computes the edge relation of the provenance graph and then translates this program into SQL.

Provenance Export. This pipeline extends Fig. 1a with an additional step that translates the relational provenance encoding of a query produced by this pipeline into PROV-JSON, which is the JSON serialization of the WC3 recommended provenance exchange format. This method [21] uses additional complex projections on top of the query instrumented for provenance capture to construct JSON document fragments and concatenate them into a single PROV-JSON document stored as a relation with a single attribute and single tuple.

B. Performance Bottlenecks of Instrumentation

While instrumentation enables diverse provenance features to be implemented on top of databases without the need to modify the DBMS itself, the performance of generated queries is often far from optimal. Based on our extensive experience with instrumentation systems [19], [21], [7], [5], [12] and a preliminary evaluation we have identified bad plan choices by the DBMS backend as a major bottleneck. Since relational optimizers have to balance time spend on optimization versus improvement of query performance, optimizations that do not benefit common workloads are typically not considered. Thus, most optimizers are incapable of simplifying instrumented queries, will not explore relevant parts of the plan space, or will spend excessive time on optimization. We now give a brief overview of problems that we have encountered in this space.

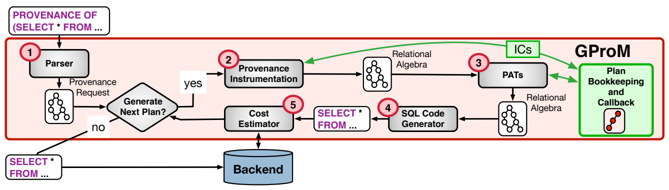


Fig. 3: GProM with Çost-based Optimizer

P1. Blow-up in Expression Size. The instrumentation for transaction provenance [5], [6] shown in Fig. 1b may produce queries with a large number of query blocks. This can lead to long optimization times in systems that unconditionally pull-up subqueries (such as Postgres) because the subquery pull-up would result in `SELECT` clause expressions of size exponential in the number of stacked query blocks. While more advanced optimizers do not apply the subquery pull-up transformation unconditionally, they will at least consider it leading to the same blow-up in expression size during optimization.

P2. Common Subexpressions. The Datalog provenance pipeline (Fig. 1c) instruments the input program using so-called firing rules to capture rule derivations. Compiling such queries into relational algebra leads to algebra graphs with many common subexpressions and a large number of duplicate elimination operators. The provenance export instrumentation mentioned above constructs the PROV output using multiple projections over an instrumented subquery that captures provenance. The large number of common subexpressions in both cases may result in very long optimization time. Furthermore, if subexpressions are not reused then this significantly increases the query size. For the Datalog queries, the choice of when to remove duplicates significantly impacts performance.

P3. Blocking Join Reordering. Provenance instrumentation as implemented in GProM [7] is based on rewrite rules. For instance, provenance annotations are propagated through an aggregation by joining the aggregation with the provenance instrumented version of the aggregation's input on the group by attributes. Such transformations increase a query's size and lead to interleaving of joins with operators such as aggregation or duplicate elimination. This interleaving may block optimizers from reordering joins leading to suboptimal join orders.

P4. Redundant Computations. Most provenance approaches instrument a query to capture provenance one operator at a time using operator-specific rewrite rules (e.g., the rewrite rules used by Perm [12]). To be able to apply such operator-specific rules to rewrite a complex query, the rules have to be generic enough to be applicable no matter how operators are combined by the query. In some cases that may lead to redundant computations, e.g., an instrumented operator generates a new column that is not needed by any downstream operators.

II. Solution Overview

We address the performance bottlenecks of instrumentation by developing heuristic and cost-based optimization techniques. While optimization has been recognized as an important problem in provenance management, previous work has almost exclusively focused on how to compress provenance to reduce storage cost, e.g., see [4], [9], [24]. In contrast, in this work we assume that the provenance encoding is given, i.e., the user requests a particular type of provenance, and study the orthogonal problem of **improving the performance of instrumented queries** that compute provenance.

We now give a brief overview of our solution and contributions. An important advantage of our approach is that it applies to any database backend and instrumentation pipeline. New transformation rules and cost-based choices can be added with ease. We implement these optimizations in GProM [7] (see Fig. 3), our provenance middleware that supports multiple DBMS backends (available as open source at https://github.com/IITDBGroup/gprom). Our optimizations which are applied during the compilation of a provenance request into SQL on average improve performance by over 4 orders of magnitude compared to unoptimized instrumented queries. When optimizing instrumented queries, we can target any of the query languages used within the pipeline, e.g., if relational algebra is the output language for a compilation step then we can apply equivalence preserving transformations to the generated algebra expression before passing it on to the next stage of the pipeline. In fact, we develop several *provenance-specific algebraic transformations* (or *PAT*s for short). In addition, we can optimize during a compilation step, i.e., if we know two equivalent ways of translating an annotated algebra operator into standard relational algebra, we should choose the one which results in a better plan. We call such decisions *instrumentation choices* (*ICs*). We developed an effective set of PATs and ICs as our first major contribution.

PATs. We identify algebraic equivalences which are usually not applied by databases, but are effective for speeding up provenance computations. For instance, we factor references to attributes to enable merging of projections without blow-off in expression size, pull up projections that create provenance annotations, and remove unnecessary duplicate elimination and window operators. Following the approach presented in [15] we infer local and non-local properties such as candidate keys for the algebra operators of a query. This enables us to define transformations that rely on non-local information.

ICs. We introduce two ways for instrumenting an aggregation operator for provenance capture: 1) using a *join* (this rule is used by Perm [12]) to combine the aggregation with the provenance of the aggregation's input; 2) using *window* functions (SQL `OVER` clause) to directly compute the aggregation functions over inputs annotated with provenance. We also present two ways for pruning tuples that are not in the provenance early-on when computing the provenance of a transaction [5].

CBO for Instrumentation. Some PATs are not always beneficial and for some ICs there is no clearly superior choice. Thus, there is a need for *cost-based optimization* (CBO). Our second contribution is a CBO framework for instrumentation pipelines. Our CBO algorithm can be applied to any such pipeline no matter what compilation steps and intermediate languages are used. This is made possible by decoupling the plan space exploration from actual plan generation.

Our optimizer treats the instrumentation pipeline as a blackbox which it calls repeatedly to produce SQL queries (**plans**). Each such plan is sent to the backend database for planning and cost estimation. We refer to an execution of the pipeline as an **iteration**. It is the responsibility of the

Operator	Definition
σ	$\sigma_\theta(R) = \{t^n \mid t^n \in R \wedge t \models \theta\}$
Π	$\Pi_A(R) = \{t^n \mid n = \sum_{u.A=t} R(u)\}$
$\cup$	$R \cup S = \{t^{n+m} \mid t^n \in R \wedge t^m \in S\}$
$\cap$	$R \cap S = \{t^{min(n,m)} \mid t^n \in R \wedge t^m \in S\}$
$-$	$R - S = \{t^{max(n-m,0)} \mid t^n \in R \wedge t^m \in S\}$
$\times$	$R \times S = \{(t,s)^{n*m} \mid t^n \in R \wedge s^m \in S\}$
γ	${}_G\gamma_{f(a)}(R) = \{(t.G, f(G_t))^1 \mid t \in R\}$ $G_t = \{(t_1.a)^n \mid {t_1}^n \in R \wedge t_1.G = t.G\}$
δ	$\delta(R) = \{t^1 \mid t \in R\}$
ω	$\omega_{f(a) \to x, G \Vert O}(R) \equiv \{(t, f(P_t))^n \mid t^n \in R\}$ $P_t = \{(t_1.a)^n \mid t_1^n \in R \wedge t_1.G = t.G \wedge t_1 \leq_O t\}$

TABLE I: Relational algebra operators

pipeline's components to signal to the optimizer the existence of optimization choices (called **choice points**) through the optimizer's **callback API**. The optimizer responds to a call from one of these components by instructing it which of the available **options** to choose. We keep track of which choices had to be made, which options exist for each choice point, and which options were chosen. This information is sufficient to iteratively enumerate the plan space by making different choices during each iteration. Our approach provides great flexibility in terms of supported optimization decisions, e.g., we can choose whether to apply a PAT or select which ICs to use. Adding a new optimization choice only requires adding a few LOC to the instrumentation pipeline to inform the optimizer about the availability of options. While our approach (Fig. 3) has some aspects in common with cost-based query transformation [2], it is to the best of our knowledge the first one that is **plan space and query language agonistic**. Since costing a plan requires us to use the DBMS to optimize a query, the number of iterations that can be run within reasonable time is limited. In addition to randomized search techniques, we also support an approach that balances optimization vs. execution time, i.e., it stops optimization once a "good enough" plan has been found.

Our approach peacefully coexists with the DBMS optimizer. We use the DBMS optimizer where it is effective (e.g., join reordering) and use our optimizer to address the database's shortcomings with respect to provenance computations. To maintain the advantage of database independence, we implement PATs and ICs in a middleware, but these optimizations could also be implemented as an extension of a regular database optimizer (e.g., as cost-based transformations [2]).

III. Background and Notation

A database schema $\mathbf{D} = \{\mathbf{R_1}, \ldots, \mathbf{R_n}\}$ is a set of relation schemas $\mathbf{R_1}$ to $\mathbf{R_n}$. A relation schema $\mathbf{R}(a_1, \ldots, a_n)$ consists of a name ($\mathbf{R}$) and a list of attribute names a_1 to a_n. The arity of a relation schema is the number of attributes in the schema. Here we use the bag-semantics version of the relational model. Let $\mathcal{U}$ be a domain of values. An instance R of an n-ary relation schema $\mathbf{R}$ is a function $\mathcal{U}^n \to \mathbb{N}$ with finite support $\mid \{t \mid R(t) \neq 0\} \mid$. We use $t^m \in R$ to denote that tuple t occurs with multiplicity m, i.e., $R(t) = m$ and $t \in R$ to denote that $R(t) > 0$. An n-ary relation R is contained in another n-ary relation S iff $\forall t \in \mathcal{U}^n : R(t) \leq S(t)$, i.e., each tuple in R appears in S with the same or higher multiplicity. We abuse notation and write $R \subseteq S$ to denote that R is contained in S.

Table I shows the definition of the bag-semantics version of relational algebra we use in this work. In addition to set operators, selection, projection, crossproduct, duplicate elimination, and join, we also support aggregation and windowed aggregation. Aggregation ${}_{G}\gamma_{f(a)}(R)$ groups tuples according to their values in attributes G and computes the aggregation function f over the values of attribute a for each group. Window operator $\omega_{f(a)\rightarrow x,G\|O}(R)$ applies function f to the window generated by partitioning the input on expressions G and ordering tuples by O. For each input tuple t, the window operator returns t with an additional attribute x storing the result of the window function. We use $Q(I)$ to denote the result of query Q over database instance I. We use $op_1 \overset{*}{\rightsquigarrow} op_2$ to denote that operator op_2 is an ancestor of (downstream of) op_1. Furthermore, we use $Q[Q_1 \leftarrow Q_2]$ to denote the substitution of subexpression Q_1 in Q with Q_2 and $\textsc{Sch}(Q)$ to denote the schema of the result of an algebra expression Q.

IV. PAT

We now introduce equivalence-preserving PAT rules that address some of the problems mentioned in Sec. I-B. These results are applied after the compilation of a provenance request into relational algebra. We present each rule as $\frac{pre}{q \rightarrow q'}$ which has to be read as "If condition pre holds, then q can be rewritten as q'". Similar to Grust et al. [15], we infer properties for the operators of an algebra expression and use these properties in preconditions of rules. These properties provide us with essential local (e.g., candidate keys) and non-local information (e.g., which attributes of the operator's result are necessary for evaluating downstream operators).

A. Operator Properties

set. Boolean property *set* denotes if an ancestor of an operator *op* is a duplicate elimination operator and the number of duplicates is irrelevant for computing operators on the path from *op* to the next duplicate elimination operator. We use *set* to remove or add duplicate elimination operators.

Definition 1. *Let op be an operator in a query Q.* $set(op) = true$ *iff* $\exists op' : op \overset{*}{\rightsquigarrow} op'$ *with* $op' = \delta$ *and* $\forall op'' : op \overset{*}{\rightsquigarrow} op'' \overset{*}{\rightsquigarrow} op'$ *we have* $op'' \notin \{\gamma, \omega\}$.

keys. Keys are the candidate keys of an operator's output. For example, consider a relation $R(a, b, c, d)$ where $\{a\}$ and $\{b, c\}$ are unique, then $keys(R) = \{\{a\}, \{b, c\}\}$.

Definition 2. *Let* $Q = op(Q')$ *be a query. A set* $E \subseteq \textsc{Sch}(Q)$ *is a* super key *for op iff for every instance I we have* $\forall t, t' \in Q(I) : t.E = t'.E \rightarrow t = t'$ *and* $t^n \in Q(I) \rightarrow n \leq 1$. *A super key E is a* candidate key *if no* $E' \subset E$ *is a super key. We use* $keys(op)$ *to denote the set of all candidate keys for op.*

EC. The equivalence class (EC) property records which attributes in the output of an operator are guaranteed to have the same value. For example, if $EC(R) = \{\{a\}, \{b, c\}, \{d\}\}$ then we know that $t.b = t.c$ for each tuple $t \in R$.

Definition 3. *Let* $Q = op(Q')$ *be a query. A set of attributes* $E \subseteq \textsc{Sch}(Q)$ *is an* equivalence class *(EC) for op iff for all instances I we have* $\forall t \in Q(I) : \forall a, b \in E : t.a = t.b$. *Equivalence classes may also contain a constant c. In this case we additionally require that* $\forall t \in Q(I) : \forall a \in E : t.a = c$.

icols. This property records which attributes are needed to evaluate ancestors of an operator. For example, attribute d in $\Pi_a(\Pi_{a,b+c\rightarrow d}(R))$ is not needed to evaluate Π_a.

Definition 4. *Let Q be a query and* $Q_{sub} = op(Q'_{sub})$ *be a subquery of Q,* $icols(op)$ *is the minimal set of attributes* $E \subseteq \textsc{Sch}(Q_{sub})$ *such that* $Q \equiv Q[Q_{sub} \leftarrow \Pi_{icols(op)}(Q_{sub})]$.

B. Property Inference

We infer properties for operators through traversals of the algebra graph. During a bottom-up traversal the property P for an operator *op* is computed based on the values of P for the operator's children. Conversely, during a top-down traversal the property P of an operator *op* is computed based on the values of P for the parents of *op*. We use $\circledast$ to denote the root operator. In the following, we show the inference of property EC in Table II and III, the inference of property $icols$ in Table IV, the inference of property set in Table V and the inference of property key in Table VI.

Inferring the EC Property. We compute EC in a bottom-up traversal followed by a top-down traversal. Table II shows the inference rules for the bottom-up traversal. In the inference rules we use an operator $\mathcal{E}^*$ that takes a set of ECs as input and merges classes if they are overlapping. This corresponds to repeated application of transitivity: $a = b \wedge b = c \rightarrow a = c$. Formally, operator $\mathcal{E}^*$ is defined as the least fixed-point of operator $\mathcal{E}$ shown below:

$$\begin{aligned}\mathcal{E}(EC) = &\{E \cup E' \mid E \in EC \wedge E' \in EC \wedge E \cap E' \neq \emptyset \wedge E \neq E'\} \\ &\cup \{E \mid E \in EC \wedge \nexists E' \in EC : E \cap E' \neq \emptyset\}\end{aligned}$$

Selection. We assume that selection conditions have been translated into conjunctive normal form $\theta_1 \wedge ... \wedge \theta_n$. If θ_i is an equality comparison $a = b$ where both a and b are attributes of relation R, then we need to merge their ECs (we denote the EC containing attribute a as $EC(R, a)$). If b is a constant, then we need to add b to $EC(R, a)$. This is realized by adding $\{a, b\}$ to the input set of ECs and merging overlapping ECs using $\mathcal{E}^*$. For example, if $\mathbf{R} = (a, b, c)$ and $EC(R) = \{\{a, b\}, \{c\}\}$, then $EC(\sigma_{a=5 \wedge c<9}(R)) = \{\{a, b, 5\}, \{c\}\}$.

Join. For each tuple t in the result of a join $R \bowtie_{a=b} S$, we know that $t.a = t.b$, because otherwise the tuple cannot be in the result. Thus, we merge $EC(R, a)$ with $EC(S, b)$ using the approach presented above (adding $\{a, b\}$). For example, if $EC(R) = \{\{a, b\}, \{c\}\}$ and $EC(S) = \{\{d\}, \{e, f\}\}$. Then, $EC(R \bowtie_{a=d} S) = \{\{a, b, d\}, \{c\}, \{e, f\}\}$.

C. Provenance-specific Transformations Rules

We now introduce the subset of our PAT rules shown in Fig. 4, prove their correctness, and then discuss how these rules address the performance bottlenecks discussed in Sec. I-B.

Provenance Projection Pull Up. Provenance instrumentation [7], [12] seeds provenance annotations by duplicating attributes of the input using projection. This increases the size of tuples in intermediate results. We can delay this duplication of attributes if the attribute we are replicating is still available in ancestors of the projection. In Rule (1), b is an attribute storing provenance generated by duplicating attribute a. If a is available in the schema of $\diamond(\Pi_A(R))$ ($\diamond$ can be any operator) and b is not needed to compute $\diamond$, then we can pull the projection on $a \rightarrow b$ through operator $\diamond$. For example,

TABLE II: Bottom-up inference of property *EC* (*Equivalence Class*) of operator $\diamond$

Operator $\diamond$	Inferred property for the input(s) of $\diamond$
R	$\{\{a\} \mid a \in \textsc{Sch}(R)\}$
$\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R)$	$\mathcal{E}^*(EC(R) \cup \{\{a, b\} \mid \exists i : \theta_i = (a = b)\})$
$\Pi_{a_1 \to b_1, \ldots, a_n \to b_n}(R)$	$\mathcal{E}^*(\{\{b_i, b_j\} \mid \exists E \in EC(R) \wedge a_i \in E \wedge a_j \in E\} \cup \{\{b_i\} \mid i \in \{1, \ldots, n\}\})$
$R \bowtie_{a=b} S$	$\mathcal{E}^*(EC(R) \cup EC(S) \cup \{\{a, b\}\})$
$R \times S$	$EC(R) \cup EC(S)$
${}_{b_1,\ldots,b_n}\gamma_{F(a)}(R)$	$\{\{b_1, \ldots, b_n\} \cap E \mid E \in EC(R)\} \cup \{\{F(a)\}\}$
$\delta(R)$	$EC(R)$
$R \cup S$	$\mathcal{E}^*(\{E \cap E' \mid E \in EC(R) \wedge E' \in EC(S)[\textsc{Sch}(S)/\textsc{Sch}(R)]\}$
$R \cap S$	$\mathcal{E}^*(EC(R) \cup EC(S)[\textsc{Sch}(S)/\textsc{Sch}(R)])$
$R - S$	$EC(R)$

TABLE III: Top-down inference of property *EC* (*Equivalence Class*) for the input(s) of operator $\diamond$

Operator $\diamond$	Inferred property *set* of input(s) of $\diamond$
$\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R)$	$EC(R) = EC(\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R))$
$\Pi_{a_1 \to b_1, \ldots, a_n \to b_n(R)}$	$EC(R) = \mathcal{E}^*(\{\{a_i, a_j\} \mid \exists E \in EC(\Pi_{a_1 \to b_1, \ldots, a_n \to b_n}(R)) \wedge b_i \in E \wedge b_j \in E\} \cup EC(R))$
$R \bowtie_{a=b} S$	$EC(R) = \{E - \textsc{Sch}(S) \mid E \in EC(R \bowtie_{a=b} S)\}$ $EC(S) = \{E - \textsc{Sch}(R) \mid E \in EC(R \bowtie_{a=b} S)\}$
$R \times S$	$EC(R) = \{E - \textsc{Sch}(S) \mid E \in EC(R \bowtie_{a=b} S)\}$ $EC(S) = \{E - \textsc{Sch}(R) \mid E \in EC(R \bowtie_{a=b} S)\}$
${}_{b_1,\ldots,b_n}\gamma_{F(a)}(R)$	$EC(R) = \mathcal{E}^*(\{E \cap \textsc{Sch}(R) \mid E \in_{b_1,\ldots,b_n} \gamma_{F(a)}(R)\} \cup EC(R))$
$\delta(R)$	$EC(R) = EC(\delta(R))$
$R \cup S$	$EC(R) = \mathcal{E}^*(EC(R \cup S) \cup EC(R))$ $EC(S) = \mathcal{E}^*(EC(R \cup S)[\textsc{Sch}(R)/\textsc{Sch}(S)] \cup EC(S))$
$R \cap S$	$EC(R) = EC(R \cap S)$ $EC(S) = EC(R \cap S)[\textsc{Sch}(R)/\textsc{Sch}(S)]$
$R - S$	$EC(R) = EC(R - S)$ $EC(S) = EC(R - S)[\textsc{Sch}(R)/\textsc{Sch}(S)]$

$$\frac{a \subseteq \textsc{Sch}(\diamond(\Pi_A(R))) \wedge b \notin icols(\diamond(\Pi_A(R)))}{\diamond(\Pi_{A, a \to b}(R)) \to \Pi_{\textsc{Sch}(\diamond(\Pi_A(R))), a \to b}(\diamond(\Pi_A(R)))} \quad (1) \qquad \frac{keys(R) \neq \emptyset}{\delta(R) \to R} \quad (2) \qquad \frac{set(\delta(R))}{\delta(R) \to R} \quad (3)$$

$$\frac{A = icols(R)}{R \to \Pi_A(R)} \quad (4) \qquad \frac{x \notin icols(\omega_{f(a) \to x}(R))}{\omega_{f(a) \to x}(R) \to R} \quad (5) \qquad \frac{e_1 = if\ \theta\ then\ A + c\ else\ A}{\Pi_{e_1, \ldots, e_m}(R) \to \Pi_{A+\ if\ \theta\ then\ c\ else\ 0, e_2, \ldots, e_m}(R)} \quad (6)$$

Fig. 4: Provenance-specific transformation (PAT) rules

TABLE IV: Top-down inference of property *icols* for the input(s) of operator $\diamond$

Operator $\diamond$	Inferred property *set* of input(s) of $\diamond$
$\circledast(R)$	$R.icols = \{a \mid a \in \textsc{Sch}(\diamond(R))\}$
$\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R)$	$R.icols = icols \cup cols(\theta_i \wedge \ldots \wedge \theta_n)$
$\Pi_{a_1 \to b_1, \ldots, a_n \to b_n(R)}$	$e.icols = \{a_1, \ldots, a_n\}$
$R \bowtie_{a=b} S$	$R.icols = (icols \cup \{a, b\}) \cap \textsc{Sch}(R)$ $S.icols = (icols \cup \{a, b\}) \cap \textsc{Sch}(S)$
$R \times S$	$R.icols = icols \cap \textsc{Sch}(R)$ $S.icols = icols \cap \textsc{Sch}(S)$
${}_{b_1,\ldots,b_n}\gamma_{F(a)}(R)$	$R.icols = icols \cup \{b_1, \ldots, b_n, a\}$
$\delta(R)$	$R.icols = \{a \mid a \in \textsc{Sch}(R)\}$
$R \cup S$	$R.icols = icols$ $S.icols = icols[\textsc{Sch}(R)/\textsc{Sch}(S)]$
$R \cap S$	$R.icols = icols$ $S.icols = icols[\textsc{Sch}(R)/\textsc{Sch}(S)]$
$R - S$	$R.icols = icols$ $S.icols = icols[\textsc{Sch}(R)/\textsc{Sch}(S)]$

TABLE V: Top-down inference of Boolean property *set* for the input(s) of operator $\diamond$

Operator $\diamond$	Inferred property *set* of input(s) of $\diamond$
$\circledast(R)$	$R.set = false$
$\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R)$	$R.set = R.set \wedge set$
$\Pi_{a_1 \to b_1, \ldots, a_n \to b_n(R)}$	$R.set = R.set \wedge set$
$R \bowtie_{a=b} S$	$R.set = R.set \wedge set$ $S.set = S.set \wedge set$
$R \times S$	$R.set = R.set \wedge set$ $S.set = S.set \wedge set$
${}_{b_1,\ldots,b_n}\gamma_{F(a)}(R)$	$R.set = R.set \wedge true$
$\delta(R)$	$R.set = R.set \wedge true$
$R \cup S$	$R.set = R.set \wedge set$ $S.set = S.set \wedge set$
$R \cap S$	$R.set = R.set \wedge set$ $S.set = S.set \wedge set$
$R - S$	$R.set = R.set \wedge set$ $S.set = S.set \wedge set$

consider a query $Q = \sigma_{a<5}(R)$ over relation $R(a, b)$. Provenance instrumentation yields: $\sigma_{a<5}(\Pi_{a,b,a \to P(a), b \to P(b)}(R))$. This projection can be pulled up to reduce the size of the selection's result tuples: $\Pi_{a,b,a \to P(a), b \to P(b)}(\sigma_{a<5}(R))$.

Remove Duplicate Elimination. Rules (2) and (3) remove duplicate elimination operators. If a relation R has at least one candidate key, then it cannot contain any duplicates. Thus, a duplicate elimination applied to R can be safely removed (Rule (2)). Furthermore, if the output of a duplicate elimination *op* is again subjected to duplicate elimination further downstream and the operators on the path between these two operators are not sensitive to the number of duplicates (property *set* is true), then *op* can be removed (Rule (3)).

Remove Redundant Attributes. Recall that $icols(R)$ are the attributes of relation R which are needed to evaluate ancestors of R in the query. If $icols(R) = A$, then we use Rule (4)

TABLE VI: Bottom-up inference of property *key* for the input(s) of operator $\diamond$

Operator $\diamond$	Inferred property *set* of input(s) of $\diamond$
$\sigma_{(\theta_i \wedge \ldots \wedge \theta_n)}(R)$	$key = R.key$
$\Pi_{a_1 \rightarrow b_1, \ldots, a_n \rightarrow b_n(R)}$	$key = \{E \mid E \in R.key \wedge \forall x \in E \text{ s.t. } x \in \{a_1, \ldots, a_n\}\}$
$R \bowtie_{a=b} S$	$key = R.key \cup S.key$
$R \times S$	$key = R.key \cup S.key$
${}_{b_1, \ldots, b_n}\gamma_{F(a)}(R)$	$key = \{E \mid E \in \{b_1, \ldots, b_n, F(a)\} \cap x, x \in R.key\}$
$\delta(R)$	$key = R.key(R.key \neq null) \cup \textsc{Sch}(R)(R.key = null)$
$R \cup S$	$key = null$
$R \cap S$	$key = R.key \cup S.key[\textsc{Sch}(R)/\textsc{Sch}(S)]$
$R - S$	$key = R.key$

to remove all other attributes by projecting R on A. For example, if $icols(R) = \{a, b\}$, then $R \rightarrow \Pi_{a,b}(R)$ is a valid rewrite. Operator $\omega_{f(a) \rightarrow x}(R)$ extends each tuple $t \in R$ by adding a new attribute x that stores the result of window function $f(a)$. Rule (5) removes ω if x is not needed by any ancestor of $\omega(R)$. For example, if $\textsc{Sch}(R) = \{a, b\}$, then $\textsc{Sch}(\omega_{sum(b) \rightarrow x}(R)) = \{a, b, x\}$. If $icols(\omega_{sum(b) \rightarrow x}(R)) = \{a, b\}$, then we can remove the window operator.

Attribute Factoring. Attribute factoring restructures projection expressions in such a way that adjacent projections can be merged without blow-up in expression size. For instance, when projections $\Pi_{b+b+b \rightarrow c}(\Pi_{a+a+a \rightarrow b}(R))$ are merged, this increases the number of references to a to 9 (each mention of b is replaced with $a + a + a$). This blow-up can occur when computing the provenance of transactions where multiple levels of `CASE` expressions are used. In relational algebra we represent `CASE` as $if\ \theta\ then\ e_1\ else\ e_2$. Rule (6) addresses a common case that arises when reenacting an update [5]. For example, update `UPDATE R SET a = a + 2 WHERE b = 2` would be expressed as $\Pi_{if\ b=2\ then\ a+2\ else\ a, b}(R)$ which can be rewritten as $\Pi_{a + if\ b=2\ then\ 2\ else\ 0, b}(R)$. Note how the number of references to attribute a was reduced from 2 to 1. We define analog rules for any arithmetic operation which has a neutral element (e.g., multiplication).

D. Addressing Instrumentation Bottlenecks through PATs

Rule (6) is a preprocessing step that helps us to avoid a blow-up in expression size when merging projections (Sec. I-B **P1**). Rules (2) and (3) can be used to remove unnecessary duplicate elimination operators (**P2**). Bottleneck **P3** is addressed by removing operators that block join reordering: Rules (2), (3), and (5) remove such operators. Even if such operators cannot be removed, Rule (1) and Rule (4) remove attributes that are not needed which reduces the schema size of intermediate results. **P4** can be addressed by using Rules (2), (3), and (5) to remove redundant operators. Furthermore, Rule (4) removes columns that are not needed. In addition to the rules discussed so far, we apply standard equivalences, because our transformations often benefit from these equivalences and they also allow us to further simplify a query. For instance, we apply *selection move-around* (which benefits from the EQ property), merging of selections and projections (only if this does not result in a significant increase in expression size), and remove redundant projections (projections on all input attributes).

E. Correctness

Theorem 1. *The PATs from Fig. 4 are equivalence preserving.*

Proof: Rule (1): The value of attribute b is the same as the value of a (follows from $a \rightarrow b$). Since b is not needed to evaluate $\diamond(\Pi_A(R))$, we can delay the computation b after $\diamond$ has been evaluated. Rule 2: Since $keys(R) \neq \emptyset$, by Def. 2 it follows that no duplicate tuples exist in R ($t^n \in R \rightarrow n \leq 1$). Thus, we get $\delta(R) \rightarrow R$. Rule 3: We say an operator $\diamond(R)$ is insensitive to duplicates if for all R we have $t \in \diamond(\delta(R)) \leftrightarrow t \in \diamond(R)$. That is, which tuples are returned by the operator is independent of input tuple multiplicities. Since $set(\delta(R))$ = true, we know that there exists $op' = \delta$ with $op \overset{*}{\rightsquigarrow} op'$ and $\forall op'' : \delta(R) \overset{*}{\rightsquigarrow} op'' \overset{*}{\rightsquigarrow} op'$ such that $op'' \notin \{\gamma, \omega\}$. Note that operator types other than γ and ω are insensitive to duplicates. Thus, the set of tuples in the input of op' is not affected by the rewrite $\delta(R) \rightarrow R$. While the multiplicities of these tuples can be affected by the rewrite, the final result of $op' = \delta$ is not affected. Rule 4: Suppose $A = icols(R)$, by definition 4 we get $R \rightarrow \Pi_A(R)$. Rule 5: From $x \notin icols(\omega_{f(a) \rightarrow x}(R))$ follows $Q[\omega_{f(a) \rightarrow x}(R) \leftarrow \Pi_{\textsc{Sch}(R)}(\omega_{f(a) \rightarrow x}(R))] \equiv Q$. Based on the definition of ω it follows that $t^n \in \Pi_{\textsc{Sch}(R)}(\omega_{f(a) \rightarrow x}(R)) \leftrightarrow t^n \in R$. Thus, $Q[\omega_{f(a) \rightarrow x}(R) \leftarrow R] \equiv Q$. Rule (6): Let $e_1' = A +$ if θ then c else 0. We distinguish two cases if θ holds, then both e_1 and e_1' evaluate to $A + c$. If $\neg\theta$ holds, then both e_1 and e_1' evaluate to A. ■

V. Instrumentation Choices

Window vs. Join. The *Join* method for instrumenting an aggregation operator for provenance capture was first used by Perm [12]. To propagate provenance from the input of the aggregation to produce results annotated with provenance, the original aggregation is computed and then joined with the provenance of the aggregations input on the group-by attributes. This will match the aggregation result for a group with the provenance of tuples in the input of the aggregation that belong to that group (see [12] for details). For instance, ${}_b\gamma_{sum(a)}(R)$ with $\mathbf{R} = (a, b)$ would be rewritten into $\Pi_{b, sum(a), P(a), P(b)}({}_b\gamma_{sum(a)}(R) \bowtie_{b=b'} \Pi_{b \rightarrow b', a \rightarrow P(a), b \rightarrow P(b)}(R))$. Alternatively, the aggregation can be computed over the input with provenance using the window operator ω by turning the group-by into a partition-by. The rewritten expression is $\Pi_{b, sum(a), P(a), P(b)}({}_{b\|}\omega_{sum(a)}(R)(\Pi_{b \rightarrow b', a \rightarrow P(a), b \rightarrow P(b)}(R)))$. The *Window* method has the advantage that no additional joins are introduced. However, as we will show in Sec. VIII, the *Join* method is superior in some cases and, thus, the choice between these alternatives should be cost-based.

FilterUpdated vs. HistJoin. Our approach for capturing the provenance of a transaction T [5], [6] only returns the provenance of tuples that were affected by T. We consider

Algorithm 1 CBO

```
1:  procedure CBO($Q$)
2:      $T_{best} \leftarrow \infty$, $T_{opt} \leftarrow 0.0$
3:      while HASMOREPLANS() $\wedge$ CONTINUE() do
4:          $t_{before} \leftarrow$ CURRENTTIME()
5:          $P \leftarrow$ GENERATEPLAN($Q$)
6:          $T \leftarrow$ GETCOST($P$)
7:          if $T < T_{best}$ then
8:              $T_{best} \leftarrow T$
9:              $P_{best} \leftarrow P$
10:         GENNEXTITERCHOICES( )
11:         $t_{after} \leftarrow$ CURRENTTIME()
12:         $T_{opt} = T_{opt} + (t_{after} - t_{before})$
13:     return $P_{best}$
```

two alternatives for achieving this. The first method is called *FilterUpdated*. Consider a transaction T with n updates and let θ_i denote the condition (WHERE-clause) of the i^{th} update. Every tuple updated by the transaction has to fulfill at least one θ_i. Thus, this set of tuples can be computed by applying a selection on condition $\theta_1 \vee \ldots \vee \theta_n$ to the input of reenactment. The alternative called *HistJoin* uses time travel to determine based on the database version at transaction commit which tuples where updated by the transaction. It then joins this set of tuples with the version at transaction start to recover the original inputs of the transaction. For a detailed description see [5]. *FilterUpdated* is typically superior, because it avoids the join applied by *HistJoin*. However, for transactions with a large number of operations or complex WHERE-clause conditions, the cost of evaluating the selection condition $\theta_1 \vee \ldots \vee \theta_n$ can be higher than the cost of the join.

VI. Cost-based Optimization

CBO Algorithm. The pseudocode for our CBO algorithm is shown in Algorithm 1. The algorithm consists of a main loop that is executed until the whole plan space has been explored (function HASMOREPLANS) or until a stopping criterion has been reached (function CONTINUE). In each iteration, function GENERATEPLAN takes the output of the parser and runs it through the instrumentation pipeline (e.g, the one shown in Fig. 3) to produce an SQL query. The pipeline components inform the optimizer about choice points using function MAKECHOICE. The resulting plan P is then costed. If the cost T of plan P generated in the current iteration is less than the cost T_{best} of the best plan found so far, then P becomes the next best plan. Finally, we decide which optimization choices to make in the next iteration using function GENNEXTITERCHOICES. While on the surface our CBO algorithm resembles standard CBO, the difference lies in implementation of the GENERATEPLAN and GENNEXTITERCHOICES functions and their interaction with the optimizer's callback API described below. This API achieves decoupling of enumeration, costing, and plan space traversal order from plan construction. As will be explained further in Sec. VI-B this enables the optimizer to enumerate all plans for a blackbox instrumentation pipeline. New choices are discovered at runtime when a step in the pipeline informs the optimizer about a choice point.

Costing. Our default cost estimation implementation uses the DBMS to create an optimal execution plan for P and estimate its cost. This ensures that we get the estimated cost for the plan that would be executed by the backend instead of estimating

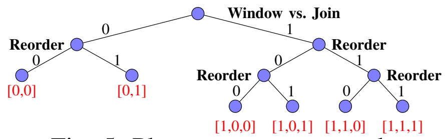


Fig. 5: Plan space tree example

cost based on the properties of the query alone. However, this is not a requirement in our framework, e.g., a separate costing module may be used for backends that do not apply CBO.

Search Strategies. Different strategies for exploring the plan space are implemented as different versions of the CONTINUE, GENNEXTITERCHOICES, and MAKECHOICE functions. The default setting guarantees that the whole search space will be explored (CONTINUE returns true). Our CBO algorithm keeps track of how much time has been spent on optimization so far (T_{opt}) which may be used to decide when to stop optimization.

A. Registering Optimization Choices

We want to make the optimizer aware of choices available in an instrumentation pipeline without having to significantly change existing code. This is achieved by registering choices though a callback interface. Thus, it is easy to introduce new choices in any step of an instrumentation pipeline at runtime by adding calls to the optimizer's MAKECHOICE function without any modifications to the optimizer and only trivial changes to the code containing the choice. The callback interface has two purposes: 1) telling the optimizer about when a choice needs to be made and how many alternatives to choose from and 2) allowing it to control which options are chosen. Recall that we refer to a point in the code where a choice is enforced as a *choice point*. A choice point has a fixed number of *options*, the optimizer's callback function returns an integer indicating to the caller which option should be taken.

Example 2. *Assume a provenance engine implements the* Join *and* Window *methods as functions* `instAggJoin` *and* `instAggWin`. *To make a cost-based choice between these methods, we call* MAKECHOICE. *The parameter passed to this function is the number n of choices ($n = 2$ in this example).*

```
if (makeChoice(2) == 0) instAggWin(Q);
else instAggJoin(Q);
```

The optimizer responds by returning a number between 0 and $n-1$ representing the choice to be taking. In our example, we would use the Window *method if 0 is returned.*

During one iteration a code fragment containing a call to MAKECHOICE may be executed several times. The optimizer treats every call as an independent choice point, e.g., 4 possible combinations of the *Join* and *Window* methods will be considered for instrumenting a query if it has two aggregations.

B. Plan Space

We now look at the shape of the search space for given a query and set of choice points. During one iteration we may hit any number of choice points and each choice made may effect what other choices have to be made in the remainder of this iteration. We use a data structure called *plan tree* that models the plan space shape. In the plan tree each intermediate node represents a choice point, outgoing edges from a node

Algorithm 2 Default MAKECHOICE Function

```
1: procedure MAKECHOICE($numChoices$)
2:     if $len(p_{next}) > 0$ then
3:         $choice \leftarrow popHead(p_{next})$
4:     else
5:         $choice \leftarrow 0$
6:     $p_{cur} \leftarrow p_{cur} \parallel choice$
7:     $n_{opts} \leftarrow n_{opts} \parallel numChoices$
8:     return choice
```

Algorithm 3 Default GENNEXTITERCHOICES Function

```
1: procedure GENNEXTITERCHOICES( )
2:     $p_{next} \leftarrow p_{cur}$
3:     for $i \in \{len(p_{next}), \ldots, 1\}$ do
4:         $c \leftarrow popTail(p_{next})$
5:         $nops \leftarrow popTail(n_{opts})$
6:         if $c + 1 < nops$ then
7:             $c \leftarrow c + 1$
8:             $p_{next} \leftarrow p_{next} \parallel c$
9:             break
10:    $p_{cur} \leftarrow []$
11:    $n_{opts} \leftarrow []$
```

are labelled with options and children represent choice points that are hit next. A path from the root of the tree to a leaf node represents a particular sequence of choices that results in the plan represented by this leaf node.

Example 3. *Assume we use the two choice points: 1) Window vs. Join; 2) reordering join inputs. The second choice point can only be hit if a join operator exist, e.g., if we choose to use the* Window *method then the resulting algebra expression may not have any joins and this choice point would never be hit. Assume we have to instrument a query which is an aggregation over the result of a join. Fig. 5 shows the corresponding plan tree. When instrumenting the aggregation, we have to decide whether to use the Window (0) or the Join method (1). If we choose (0), then we can still decide wether to reorder the inputs of the join or not. If we choose (1), then there is an additional join for which we have to decide whether to reorder its input. The tree is asymmetric, i.e., the number of choices to be made in each iteration (path in the tree) is not constant.*

C. Plan Enumeration

While the plan space tree encodes all possible plans for a given query and set of choice points, it would not be feasible to fully materialize it, because its size can be exponential in the maximum number of choice points that are hit during one iteration (the depth d of the plan tree). Our default implementation of the GENERATENEXTPLAN and MAKECHOICE functions explores the whole plan space using $\mathcal{O}(d)$ space. As long as we know the path taken in the previous iteration (represented as a list of choices as shown in Fig. 5) and for each node (choice point) on this path the number of available options, then we can determine what choices should be made in the next iteration to reach the leaf node (plan) immediately to the right of the previous iteration's plan. If p_{cur} is the path explored in the previous iteration, then by taking the next available choice as late as possible on the path will lead to the next node on the leaf level. Let p_{next} be the prefix of p_{cur} that ends in the new choice to be taken at the latest possible step. If following p_{next} leads to a path is longer than p_{next}, then after making $len(p_{next})$ choices the first option should be chosen for the remaining choice points.

Example 4. *Reconsider the plan tree shown in Fig. 5 and assume the plan created in the previous iteration is* $[0, 1]$. *The second choice point hit does have no additional options, but the first one does have one additional option. That is, to reach the next leaf node to the right of* $p_{cur} = [0, 1]$, *we should choose the other option* $p_{next} = [1]$. *This option leads us to a path that is longer than* $len(p_{next}) = 1$, *thus we choose the first option for all remaining choice points leading us to the leaf node* $[1, 0, 0]$. *In the next iteration we have one more choice available in the last step of the path leading us to* $[1, 0, 1]$. *This process continues until no more choices are available.*

We use square brackets to denote lists, e.g., $p_{next} = [0, 1]$ denotes a list with two element 0 and 1. We use $[]$ to represent an empty list. $L \leftarrow L \parallel e$ denotes appending element e to list L. Functions $popHead(L)$ and $popTail(L)$ remove and return the first (respective last) element of list L.

The makeChoice Function. Algorithm 2 shows as long as there are predetermined choices available (list p_{next}) we pick the next choice from this list. If list p_{next} is empty, then we pick the first choice (0). In both cases the choice is appended to the current path and the number of available options for the current choice point is appended to list n_{opts}.

Determining Choices for the Next Iteration. Algorithm 3 determines which options to be picked in the next iteration. We copy the path from the previous iteration (line 2) and then repetitively remove elements from the tail of the path and from the list storing the number of options (n_{ops}) until we have removed an element c for which at least one more alternative exits ($c + 1 < n_{ops}$). Once we have found such an element we append $c + 1$ as the new last element to the path.

Given a set of choice points, our algorithm is guaranteed to enumerate all plans for an input.

Theorem 2. *Let Q be input query. Algorithm 1 iterates over all plans that can be created for the given choice points.*

D. Alternative Search Strategies

Traversal Order. A simple solution for dealing with large search spaces is to define a threshold τ for the number of iterations and stop optimization once this threshold is reached. However, the search space traversal strategy we have introduced in Sec. VI-C (we call it *sequential-leaf-traversal* in the following) is not suited well for this solution. Since it only changes one choice at a time, the plans explored by this strategy for a plan space that is large compared to threshold τ are likely quite similar. To address this problem, we have developed a second strategy which we call *binary-search-traversal*. This strategy approximates a binary search over the leaves of a plan tree. The method maintains a queue of intervals (pairs of paths in the plan tree) initialized with the path to the left-most and right-most leaf of the plan tree. The strategy repeats the following steps until all plans have been explored: 1) fetch an interval $[P_{low}, P_{high}]$ from the queue; 2) compute a path that is approximately a prefix of the path to the leaf that lies in the middle of the interval; 3) extend this path to create a plan P_{middle}; and 4) push two new intervals to the end of the queue: $[P_{low}, P_{middle}]$ and $(P_{middle}, P_{high}]$.

Simulated Annealing. Metaheuristics such as simulated annealing and genetic algorithms have a long tradition in query optimization to deal with large search spaces, e.g., some systems apply metaheuristics for join enumeration once the number of joins exceeds a threshold or for cost-based query transformations [2]. We have implemented the *Simulated Annealing* metaheuristic. This method starts from a randomly generated plan and in each step applies a random transformation to derive a plan P_{cur} from the previous plan P_{pre} (let C_{cur} and C_{pre} denote the costs of these plans). If $C_{cur} < C_{pre}$, P_{cur} is used as P_{pre} for the next iteration. Otherwise, the choice of whether to discard P_{cur} or use it as the new P_{pre} is made probabilistically. The probability depends on the cost difference $C_{cur} - C_{pre}$ and a parameter $temp$ called the temperature which is decreased over time based on a cooling rate. Initially, the probability to choose an inferior plan is higher to avoid getting stuck in a local minima early on. By decreasing the temperature (and, thus also probability) over time, the approach will converge eventually.

Balancing Optimization vs. Runtime. All strategies discussed so far have the disadvantage that they do not adapt the effort spend on optimization based on how expensive the input query is. Obviously, spending more time on optimization than on execution is undesirable (assuming that provenance requests are typically ad hoc). Ideally, we would like to minimize the sum of the time spend on optimization (T_{opt}) and the execution time of the best plan T_{best} by stopping optimization once a cheap enough plan has been found. This is obviously an online problem, i.e., after each iteration we can decide to either execute the current best plan or continue to produce more plans with the hope to discover a better plan in future iterations. The following simple stopping condition results in a 2-competitive algorithm (i.e., $T_{opt} + T_{best}$ is guaranteed to be less than 2 times the minimal achievable cost) if the time spend in each iteration is bound by a constant: stop iteration if $T_{best} < T_{opt}$.

VII. Related Work

Our work is related to optimization techniques that sit on top of standard CBO, to other approaches for compiling non-relational languages into SQL, and to optimization of provenance capture and storage.

Cost-based Query Transformation. State-of-the-art DBMS apply transformations such as decorrelation of nested subqueries [22] in addition to (typically exhaustive) join enumeration and choice of physical operators. Often such transformations are integrated with CBO [2] by iteratively rewriting the input query through transformation rules and then finding the best plan for each rewritten query. Typically, metaheuristics (randomized search) are applied to deal with the large search space. Extensibility of query optimizers has been studied in, e.g., [13]. While our CBO framework is also applied on-top of standard database optimization, we can turn any choice (e.g., ICs) within an instrumentation pipeline into a cost-based decision while cost-based query transformation is typically limited to algebraic transformations. Furthermore, our framework has the advantage that new optimization choices can be added without modifying the optimizer and with minimal changes to existing code.

Compilation of Non-relational Languages into SQL. Approaches that compile non-relational languages (e.g., XQuery [15], [20]) or extensions of relational languages (e.g., temporal [23] and nested collection models [11]) into SQL face similar challenges as we do. Grust et al. [15] optimize compilation of XQuery into SQL. The approach heuristically applies algebraic transformations to cluster join operations with the goal to produce an SQL query that can successfully be optimized by a relational database. We adopt the idea of inferring properties over algebra graphs introduced in this work. However, to the best of our knowledge we are the first to integrate these ideas with CBO. Furthermore, we also optimize the compilation steps in an instrumentation pipeline.

Provenance Instrumentation. Several systems such as *DB-Notes* [8], *Trio* [1], *ORCHESTRA* [17], *Perm* [12], *LogicBox* [14], *ExSPAN* [25], and *GProM* [7] model provenance as annotations on data and capture provenance by propagating annotations. Most systems apply the *provenance instrumentation* approach described in the introduction by compiling provenance capture and queries into a relational query language (typically SQL). Thus, the techniques we introduce in this work are applicable to a wide range of systems.

Optimizing Provenance Computation and Storage. Optimization of provenance has mostly focused on minimizing the storage size of provenance. Chapman et al. [9] introduce several techniques for compressing provenance information, e.g., by replacing repeated elements with references and discuss how to maintain such a storage representation under updates. Similar techniques have been applied to reduce the storage size of provenance for workflows that exchange data as nested collections [4]. A cost-based framework for choosing between reference-based provenance storage (the provenance of a tuple is distributed over several nodes) and propagating full provenance (full provenance is propagated alongside the tuple) was introduced in the context of declarative networking [25]. This idea of storing just enough information to be able to reconstruct provenance through instrumented replay, has also been adopted for computing the provenance for transactions [7], [5], [6] and in the Subzero system [24]. Subzero switches between different provenance storage representations in an adaptive manner to optimize the cost of provenance queries. Amsterdamer et al. [3] demonstrate how to rewrite a UCQ query with inequalities into an equivalent query with provenance of minimal size. Our work is orthogonal to these approaches in that we try to minimize the time for on-demand provenance generation and queries over provenance instead of compressing provenance to minimize storage size. It would be interesting to integrate compact representations of provenance with CBO, e.g., choose among alternative compression methods.

VIII. Experiments

Our evaluation focuses on measuring 1) the effectiveness of CBO in choosing the most efficient ICs and PATs, 2) the effectiveness of heuristic application of PATs, 3) the overhead of heuristic and cost-based optimization, and 4) the impact of CBO search space traversal strategies on optimization and execution time. All experiments were executed on a machine with 2 AMD Opteron 4238 CPUs, 128GB RAM, and a hardware RAID with 4 $\times$ 1TB 72.K HDs in RAID 5 running commercial DBMS X (name omitted due to licensing restrictions).

To evaluate the effectiveness of our CBO vs. heuristic optimization choices, we compare the performance of instrumented queries generated by the CBO (denoted as ***Cost***) against queries generated by selecting a predetermined option for each choice point. Based on a preliminary study we have selected three choice points: 1) using the ***Window*** or ***Join*** method; 2) using ***FilterUpdated*** or ***HistJoin*** and 3) choosing whether to apply PAT rule (3) (remove duplicate elimination). If CBO is deactivated, then we always remove such operators if possible. The application of the remaining PATs introduced in Sec. IV turned out to be always beneficial in our experiments. Thus, these PATs are always applied as long as their precondition is fulfilled. We consider two variants for each of method: activating heuristic application of the remaining PATs (suffix ***Heu***) or deactivating them (***NoHeu***). Unless noted otherwise, results were averaged over 100 runs.

A. Datasets & Workloads

Datasets. TPC-H: We have generated TPC-H benchmark datasets of size 10MB, 100MB, 1GB, and 10GB (SF0.01 to SF10). Synthetic: For the transaction provenance experiments we use a 1M tuple relation with uniformly distributed numeric attributes. We vary the size of the transactional history (this affects performance, because the database has to store this history to enable time travel which is used when capturing transaction provenance). Parameter HX indicates $X\%$ of history, e.g., $H10$ represents 10% history (100K tuples). DBLP: This dataset consistes of 8 million co-author pairs extracted from DBLP (http://dblp.uni-trier.de/xml/).

Simple aggregation queries. This workloads computes the provenance of queries consisting solely of aggregation operations using the instrumentation technique based on the rewrite rules pioneered in Perm [12] and extended in GProM [7]. An aggregation query consisting of i aggregation operations where each aggregation operates on the result of the previous aggregation. The leaf operation accesses the TPC-H `part` table. Every aggregation groups the input on a range of primary key attribute values such that the last step returns the same number of results independent of i.

TPC-H queries. We have selected 11 queries from the 22 TPC-H queries to evaluate optimization of provenance computations for complex queries. We use GProM's instrumentation approach to compute provenance.

Transactions. We use the *reenactment* approach of GProM [5], [6] to compute provenance for transactions. The transactional workload is run upfront (not included in the measured execution time) and provenance is computed retroactively. We vary the number of updates per transaction, e.g., $U10$ is a transaction with 10 updates. The tuples to be updated are selected randomly using the primary key of the relation. All transactions were executed under isolation level `SERIALIZABLE`.

Provenance export. We use the approach from [21] to translate a relational encoding of provenance (see Sec. I) into PROV-JSON. We export the provenance for a query over the TPC-H schema that is a foreign key join across relations nation, customer, and orders.

Datalog provenance queries. We use the approach described in [19] using the pipeline shown in Fig. 1c. The input is a non-recursive Datalog query Q and a user question asking why (or why-not) a set of tuples is in the result of query Q. We use the DBLP co-author dataset for this experiment and the following queries. **Q1**: Returns authors which have co-authors that have co-authors. **Q2**: Returns authors that are co-authors, but not of themselves (while this is semantically meaningless, it is a way to test negation). **Q3**: Return pairs of authors that are indirect co-authors, but are not direct co-authors. **Q4**: Return start points of paths of length 3 in the co-author graph. For each query we consider multiple why questions that specify the set of results for which provenance should be generated. We use Qi.j to denote the j^{th} why question for query Qi.

B. Measuring Query Runtime

Overview. Figure 15 shows an overview of our results. We show the average runtime of each method relative to the best method per workload, e.g., if *Cost* performs best for a workload then its runtime is normalized to 1. We use relative overhead instead of total runtime over all workloads, because some of the workloads are significantly more expensive than others and, thus, comparing the results would be biased towards these workloads. For the *NoHeu* and *Heu* methods we report the performance of the best and the worst option for each choice point. For instance, for the *SimpleAgg* workload the performance is impacted by the choice of whether the *Join* or *Window* method is used to instrument aggregation operators with *Window* performing better (*Best*). Numbers prefixed by a $'+'$ indicate that for this method some queries of the workload did not finish within the maximum time we have allocated for each query. Hence, the runtime for these cases should be interpreted as a lower bound on the actual runtime. Compared with other methods, *Cost+Heu* is on average only 4% worth then the best method for the workload and has 18% overhead in the worst case. Note that we confirmed that in all cases where an inferior plan was chosen by our CBO that was because of inaccurate cost estimations by the backend database. If we heuristically choose the best option for each choice point, then this results in a 178% overhead over CBO on average. However, achieving this performance requires that the best option for each choice point is known upfront. The impact of bad choices on average increases runtime by a factor of $\sim$ 14 compared to CBO. These results also confirm the critical importance of our PATs since deactivating these transformations increases runtime by a factor of $\sim$ 1,800 on average and more than 12,000 in the worst case.

Simple Aggregation Queries. We measure the runtime of computing provenance for the *SimpleAgg* workload over the 1GB and 10GB TPC-H datasets varying the number of aggregations per query. The total workload runtime is shown in Fig. 6 (the best method is shown in bold). We also show the average runtime per query relative to the runtime of *NoHeu+Join*. Cost-based optimization significantly outperforms the other methods. The *Window* method is more effective than the *Join* method if a query contains multiple levels of aggregation. Our heuristic optimization improves the runtime of this method by about 50%. The unexpected high runtimes of *Join+Heu* are explained below. Fig. 7 and 8 show the results for individual queries. Note that the y-axis is log-scale. Activating heuristic optimizations improves performance in most cases, but for this workload the dominating factor is choosing the right method for instrumenting aggregations. The exception is the *Join* method, where runtime increases when heuristic optimization is activated. We inspected the plans used by the backend DBMS for this case. A suboptimal join order was chosen for *Join+Heu* based on inaccurate estimations of intermediate

Queries	NoHeu+Join	Heu+Join	NoHeu+Window	Heu+Window	Cost+Heu
SAgg 1G	4.79	20.21	4.38	2.69	**0.81**
SAgg 10G	44.06	524.78	42.62	27.47	**7.65**
TPC-H 1G	+173,053.17	**199.62**	173,041.27	250.18	235.79
TPC-H 10G	+175,371.02	**2,033.71**	175,530.53	2,247.39	2,196.01

Queries	NoHeu+Join	Heu+Join	NoHeu+Window	Heu+Window	Cost+Heu
SAgg 1G	1	3.927	0.946	0.600	**0.261**
SAgg 10G	1	9.148	0.984	0.655	**0.265**
TPC-H 1G	1	**0.187**	0.955	0.220	0.203
TPC-H 10G	1	0.198	0.975	0.180	**0.174**

Fig. 6: Total runtime (**Left**) and average runtime (**Right**) per query relative to NoHeu+Join for *SimpleAgg* and *TPC-H* workloads

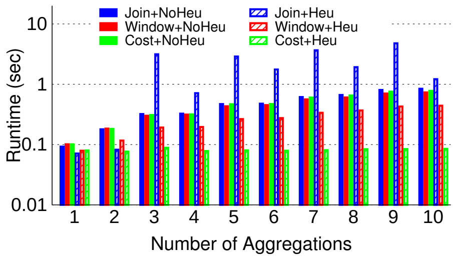


Fig. 7: 1GB *SimpleAgg* Runtime

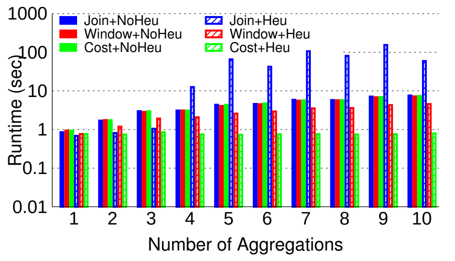


Fig. 8: 10GB *SimpleAgg* Runtime

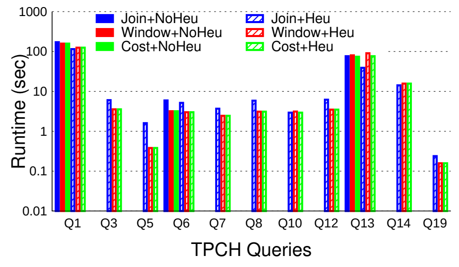


Fig. 9: Runtime *TPC-H* - 1GB

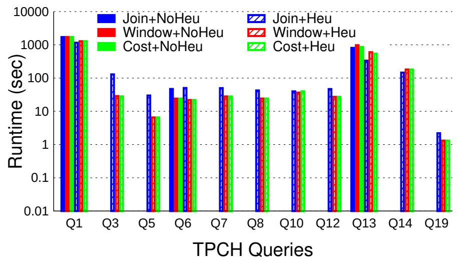


Fig. 10: Runtime *TPC-H* - 10GB

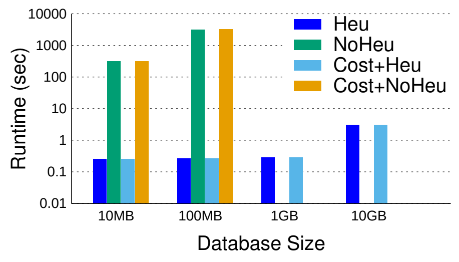


Fig. 11: Provenance Export

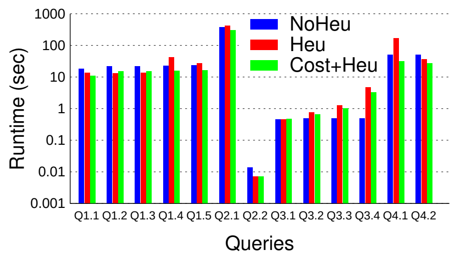


Fig. 12: Datalog Provenance

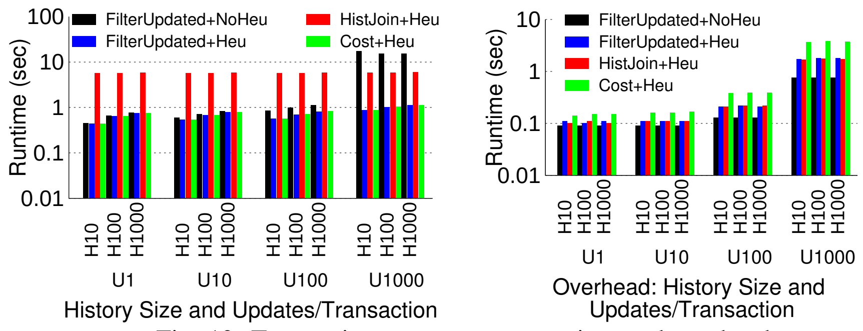


Fig. 13: Transaction provenance - runtime and overhead

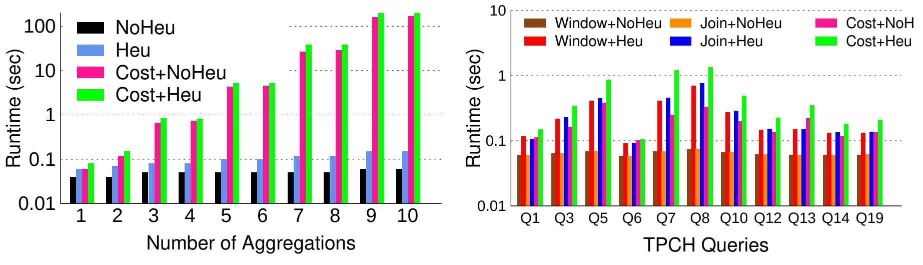


Fig. 14: *SimpleAgg* (**Left**) and *TPC-H* (**Right**) Overhead

	NoHeu (Worst)	NoHeu (Best)	Heu (Worst)	Heu (Best)	Cost+Heu
Min	1.33	1.33	**1.00**	**1.00**	**1.00**
Avg	1,878.76	1,877.95	14.16	2.82	**1.04**
Max	+12,173.35	+12,173.35	68.63	7.80	**1.18**

Fig. 15: Min, max, and avg runtime relative to the best method per workload aggregated over all workloads.

Queries Queries	FilterUpdated +NoHeu	HistJoin +Heu	FilterUpdated +Heu	Cost +Heu
HSU/T	55.11	69.50	**8.91**	**8.96**
TAPU	30.13	26.08	**12.94**	**12.89**

Fig. 16: Total workload runtime for transaction provenance

Queries	NoHeu	Heu	Cost+Heu
Export 10M	310.49	**0.25**	**0.25**
Export 100M	3,136.94	**0.27**	**0.26**
Export 1G	+21,600	**0.28**	**0.28**
Export 10G	+21,600	**3.03**	**3.01**
Datalog Provenance	583.96	736.50	**437.75**

Fig. 17: Total runtime for export and Datalog workloads

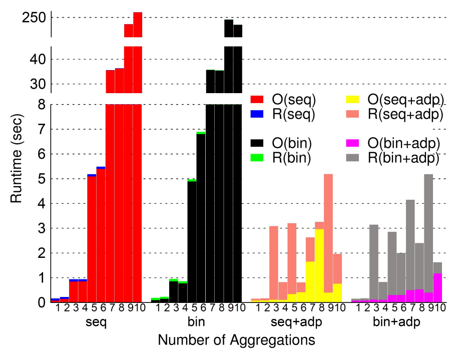


Fig. 18: Optimization + runtime for Simple Agg. - 1GB

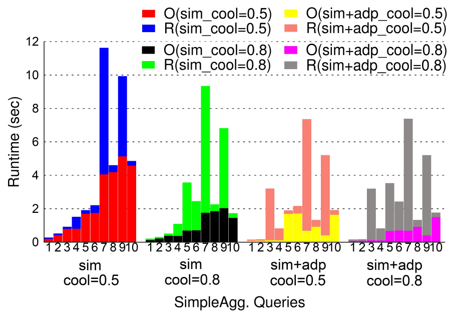


Fig. 19: Optimization + runtime for Simple Aggregation workload using Simulated Annealing - 1GB dataset

result sizes. For *Join* the DBMS did not remove intermediate operators that blocked join reordering and, thus, executed the joins in the order provided in the input query which turned out to be more efficient in this particular case. Consistently, the cost-based optimizer was either able to select *Window* as the superior method (we confirmed this by inspecting the generated execution plan) or to outperform both *Window* and *Join* by instrumenting some of the aggregations in a query using the *Window* and others with the *Join* method.

TPC-H Queries. We compute the provenance of TPC-H queries to determine whether the results for simple aggregation queries translate to more complex queries. The total workload execution time is shown in Fig. 6. We also show the average runtime per query relative to the runtime of *NoHeu+Join*. Fig. 9 and 10 show the running time for each query for the 1GB and 10GB datasets. Our CBO significantly outperforms the other methods with the only exception of *Heu+Join*. Note that the runtime of *Heu+Join* for Q13 and Q14 is lower than *Cost+Heu* which causes this effect. Depending on the dataset size and query, there are cases where the *Join* method is superior and others where the *Window* method is superior. The runtime difference between these methods is less pronounced than for *SimpleAgg* presenting a challenge for our CBO. Except for Q13 which contains 2 aggregation operators, all other queries only contain one aggregation operator. The CBO was able to determine the best method to use in almost all cases. We confirmed that for the queries where we made an inferior choice, this was based on inaccurate cost estimates. We also show the results for *NoHeu*. However, only three queries finished in the allocated time slot of 6 hours (Q1, Q6 and Q13). Thus, the TPC-H results demonstrate the need for PATs and the robustness of our CBO in being able to choose the right instrumentation for a given query.

Transactions. We next compute the provenance of transactions executed over the synthetic dataset using the techniques introduced in [5]. We vary the number of updates per transaction ($U1$ up to $U1000$) and the size of the database's history ($H10$, $H100$, and $H1000$). The total workload runtime is shown in Fig. 16. The left graph in Fig. 13 shows detailed results. We compare the runtime of *FilterUpdated* and *HistJoin* (*Heu* and *NoHeu*) with *Cost+Heu*. Our CBO choses *FilterUpdated* as the better option for this workload.

Provenance Export. Fig. 11 shows results for the provenance export workload for dataset sizes from 10MB up to 10GB (total workload runtime is shown in Fig. 17). *Cost+Heu* and *Heu* both outperform *NoHeu* demonstrating the key role of PATs for this workload. Our provenance instrumentations introduce window operators for enumerating intermediate result tuples which prevent the database from pushing selections and reordering joins. *Heu* outperforms *NoHeu*, because *Heu* determines that some of these window operators are redundant and can be removed (PAT rule (5)). The CBO does not further improve the runtime for this workload, because this export query does not contain any aggregation and duplicate elimination operators, i.e., none of the choice points were hit.

Why Questions for Datalog. The approach [19] we use for generating provenance for Datalog queries with negation may produce queries which contain a large amount of duplicate elimination operators and shared subqueries. The heuristic application of PATs would remove all but the top-most duplicate elimination operator (rules (2) and (3) in Fig. 4). However, this is not always the best option, because a duplicate elimination, while adding overhead, can reduce the size of inputs for downstream operators. Thus, as mentioned before we consider the application of rule 2 as an optimization choice in our CBO. The total workload runtime and results for individual queries are shown in Fig. 17 respective Fig. 12. Removing all redundant duplicate elimination operators (*Heu*) is not always better than removing none (*NoHeu*). Our CBO (*Cost+Heu*) has the best performance in almost all cases by choosing a subset of duplicate elimination operators to remove. Incorrect choices are again based on inaccurate cost estimation.

C. Optimization Time and CBO Strategies

Simple Aggregation. We show the optimization time of several methods in Fig. 14 (left). Heuristic optimization (*Heu*) results in an overhead of $\sim$50ms compared to the time of compiling the provenance request without any optimization (*NoHeu*) and this overhead is only slightly affected by the number of aggregations in the query. When increasing the number of aggregations, the running time of *Cost* increases more significantly because we have 2 choices for each aggregation, i.e., the plan space size is 2^i for i aggregations. We have measure where time is spend during cost-based optimization and have determined that the majority of time is spend in costing SQL queries using the backend DBMS. Note that even though we did use the exhaustive search space traversal method for our CBO, the sum of optimization time and runtime for *Cost* is still less than this sum for the *Join* method.

TPC-H Queries. In Fig. 14 (right), we show the optimization time for TPC-H queries. Activating PATs results in $\sim$50ms overhead in most cases with a maximum overhead of $\sim$0.5s. This is more than offset by the gain in query performance (recall that with *NoHeu* only 3 queries finish within 1 hour for the 1GB dataset). CBO takes up to 3s in the worst case.

CBO Strategies. We now compare query runtime and optimization time for the CBO search space traversal strategies introduced in Sec. VI. Recall that the *sequential-leaf-traversal* ***(seq)*** and *binary-search-traversal* ***(bin)*** strategies are both exhaustive strategies. *Simulated Annealing* **(sim)** is the meta-heuristic as introduced in Sec. VI-D. We also combine these strategies with our *adaptative* ***(adp)*** heuristic that limits time spend on optimization based on the expected runtime of the best plan found so far. Fig. 18 shows the total time (runtime (**R**) + optimization time (**O**)) for the simple aggregation workload. We use this workload because it contains some queries with a large plan search space. Not surprisingly, the runtime of queries produced by *seq* and *bin* is better than *seq+adp* and *bin+adp* as *seq* and *bin* traverse the whole search space. However, their total time is much higher than *seq+adp* and *bin+adp* for larger numbers of aggregations. Fig. 19 shows the total time of *sim* with and without the *adp* strategy for the same workload. We used cooling rates of 0.5 and 0.8 because they result in better performance than other rates that we have tested. The *adp* strategy improves the total runtime in all cases except for the query with 3 aggregation operators.

IX. Conclusions and Future Work

We present the first cost-based optimization framework for provenance instrumentation and its implementation in GProM. The motivation for this work is that instrumented queries which capture provenance are often not successfully optimized, even by sophisticated database optimizers. Our approach supports both heuristic and cost-based choices and is applicable to a wide range of instrumentation pipelines. We have developed several provenance-specific algebraic transformation (PAT) rules which significantly improve performance as well as study instrumentation choices (ICs), i.e., alternative ways of realizing provenance capture. Our experimental evaluation demonstrates that our optimizations improve performance by several order of magnitude for diverse provenance tasks. There are several interesting avenues of future work. We would like to improve the performance of CBO by making our optimizer aware of the structure of a query such that it can cache the best plan for a subquery. Furthermore, we plan to use the CBO to select among alternative compressed and approximate provenance representations when capturing provenance.

REFERENCES

[1] C. C. Aggarwal. Trio a system for data uncertainty and lineage. In *Managing and Mining Uncertain Data*, pages 1–35. Springer, 2009.

[2] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes. Cost-based query transformation in Oracle. In *PVLDB*, pages 1026–1036, 2006.

[3] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. *TODS*, 37(4):30, 2012.

[4] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *EDBT*, pages 958–969, 2009.

[5] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Formal foundations of reenactment and transaction provenance. Technical Report IIT/CS-DB-2016-01, IIT, 2016.

[6] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, 2016.

[7] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for database queries, updates, and transactions. *TaPP*, 2014.

[8] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB*, 14(4):373–396, 2005.

[9] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, pages 993–1006, 2008.

[10] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[11] J. Cheney, S. Lindley, and P. Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*, pages 1027–1038, 2014.

[12] B. Glavic, R. J. Miller, and G. Alonso. Using SQL for efficient generation and querying of provenance information. In *In Search of Elegance in the Theory and Practice of Computation*, pages 291–320. Springer, 2013.

[13] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[14] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[15] T. Grust, M. Mayr, and J. Rittinger. Let SQL drive the XQuery workhorse (XQuery join graph isolation). In *EDBT*, pages 147–158, 2010.

[16] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD*, 41(3):5–14, 2012.

[17] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *TODS*, 38(3):19, 2013.

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

[19] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. Implementing Unified Why- and Why-Not Provenance Through Games. In *TaPP*, 2016.

[20] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery processing in Oracle XMLDB. In *SIGMOD*, pages 828–833, 2005.

[21] X. Niu, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. Interoperability for provenance aware databases using PROV and JSON. In *TaPP*, 2015.

[22] P. Seshadri, H. Pirahesh, and T. Leung. Complex Query Decorrelation. *ICDE*, pages 450–458, 1996.

[23] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Adaptable query optimization and evaluation in temporal middleware. In *SIGMOD*, pages 127–138, 2001.

[24] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, pages 865–876, 2013.

[25] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *SIGMOD*, pages 615–626, 2010.