Heuristic and Cost-based Optimization for Diverse Provenance Tasks

Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, Venkatesh Radhakrishnan

Abstract—A well-established technique for capturing database provenance as annotations on data is to *instrument* queries to propagate such annotations. However, even sophisticated query optimizers often fail to produce efficient execution plans for instrumented queries. We develop provenance-aware optimization techniques to address this problem. Specifically, we study algebraic equivalences targeted at instrumented queries and alternative ways of instrumenting queries for provenance capture. Furthermore, we present an extensible heuristic and cost-based optimization framework utilizing these optimizations. Our experiments confirm that these optimizations are highly effective, improving performance by several orders of magnitude for diverse provenance tasks.

Index Terms—Databases, Provenance, Query Optimization, Cost-based Optimization

1 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 [1], [2] is to model provenance as annotations on data and define a query semantics 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 [3], GProM [4], DBNotes [5], LogicBlox [2], declarative Datalog debugging [6], ExSPAN [7], 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. 2a. We refer to this reduction from annotated to standard relational semantics as *provenance instrumentation* or *instrumentation* for short. The example below introduces a relational encoding of provenance polynomials [1] and the instrumentation approach for this model implemented in Perm [3].

Example 1. *Consider a query over the database in Fig. 1 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 query's result is shown in Fig. 1d. Using provenance polynomials to represent provenance, tuples in the database are annotated with variables encoding tuple identifiers (shown to the left of each tuple). Each query result is annotated with a polynomial that explains how the tuple was derived by combining input tuples. Here, addition corresponds to alternative use of tuples (e.g., union) and multiplication to conjunctive use (e.g., a join). For example, the tuple (Aldi) *is 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. Fig. 1e shows a relational encoding of these annotations as supported by the Perm [3] and GProM [4] 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 (see [3]). This encoding is computed by compiling the input query with annotated semantics into relational algebra. The resulting* instrumented *query is shown below. It adds input relation attributes 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 [3] for details).

The present paper extends [8]. An appendix with additional details is available as a supplementary document.

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

(a) sale

	name	#emp
s_1	Aldi	3
s_2	Cosco	14

(b) shop

	id	price
i_1	Steak	100
i_2	Butter	10
i_3	Bread	25

(c) item

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

(d) Provenance of Q

	result	**prov. shop**		**prov. sales**		**prov. items**	
	name	P(name)	P(#emp)	P(shop)	P(item)	P(id)	P(price)
$s_1 \cdot a_1 \cdot i_1$	Aldi	Aldi	3	Aldi	Steak	Steak	100
$s_1 \cdot a_3 \cdot i_3$	Aldi	Aldi	3	Aldi	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. 1: Provenance annotations and relational encoding

- *X.Niu, R.Kapoor and B. Glavic, Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA.*
  E-mail: {xniu7, rkapoor7}@hawk.iit.edu, bglavic@iit.edu.
- *D. Gawlick, Z.H.Liu and V. Krishnaswamy, Oracle, Rewood City, CA 94065, USA.*
  E-mail: {dieter.gawlick, zhen.liu, vasudha.krishnaswamy}@oracle.com.
- *V. Radhakrishnan, YugoByte, Sunnyvale, CA 94085 , USA.*
  E-mail: venkatesh@yugabyte.com.

1.1 Instrumentation Pipelines

In this work, we focus on optimizing instrumentation pipelines such as the one from Example 1. These pipelines divide the compilation of a frontend language to a target language into multiple compilation steps using one or more intermediate languages. We now introduce a subset of the pipelines supported by our approach to illustrate the breadth of applications supported by instrumentation. Our approach can be applied to any data management task that can be expressed as instrumentation. Notably, our implementation already supports additional pipelines, e.g., for summarizing provenance and managing uncertainty.

L1. Provenance for SQL Queries. The pipeline from Fig. 2a is applied by many provenance systems, e.g., DBNotes [5] uses L1 to compute Where-provenance [9].

L2. Provenance for Transactions. Fig. 2b shows a pipeline that retroactively captures provenance for transactions [10]. In addition to the steps from Fig. 2a, this pipeline uses a compilation step called *reenactment*. Reenactment translates transactional histories with annotated semantics into equivalent temporal queries with annotated semantics.

L3. Provenance for Datalog. This pipeline (Fig. 2c) produces provenance graphs that explain which successful and failed rule derivations of an input Datalog program are relevant for (not) deriving a (missing) query result tuple of interest [11]. A provenance request is compiled into a Datalog program that computes the edge relation of the provenance graph. This program is then translated into SQL.

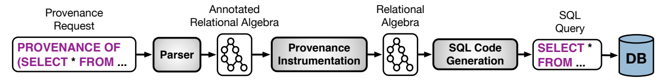
L4. Provenance Export. This pipeline (Fig.1d in Appendix A) [12] is an extension of L1 which translates the relational provenance encoding produced by L1 into PROV-JSON, the JSON serialization of the PROV provenance exchange format. This method [12] adds additional instrumentation on top of a query instrumented for provenance capture to construct a single PROV-JSON document representing the full provenance of the query. The result of L4 is an SQL query that computes this JSON document.

L5. Factorized Provenance. L5 (Fig.1e in Appendix A) captures provenance for queries. In contrast to L1, it represents the provenance polynomial of a query result as an XML document. The nested representation of provenance produced by the pipeline is factorized based on the structure of the query. The compilation target of this pipeline is SQL/XML. The generated query directly computes this factorized representation of provenance.

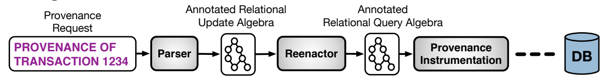
L6. Sequenced Temporal Queries. This pipeline (Fig.1f in Appendix A) translates temporal queries with sequenced semantics [13] into SQL queries over an interval encoding of temporal data. A non-temporal query evaluated over a temporal database under sequenced semantics returns a temporal relation that records how the query result changes over time (e.g., how an employee's salary changes over time). Pipeline L6 demonstrates the use of instrumentation beyond provenance. We describe Pipelines L5 and L6 in more detail in Appendix A.1 and A.2.

1.2 Performance Bottlenecks of Instrumentation

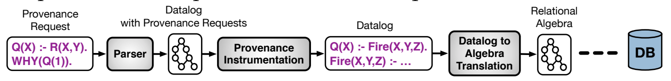
While instrumentation enables diverse provenance features to be implemented on top of DBMS, the performance of instrumented queries is often suboptimal. Based on our extensive experience with instrumentation systems [11], [12], [4], [10], [3] and a preliminary evaluation we have identified bad plan choices by the DBMS backend as a major bottleneck. Since query optimizers have to trade optimization time for 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 an overview of problems we have encountered.



(a) Provenance is captured using an annotated version of relational algebra which is first translated into relational algebra over a relational encoding of annotated relations and then 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 uses *reenactment* [10] to compile annotated updates into annotated queries.


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 provenance graphs for Datalog queries [11] based on a rewriting called *firing rules*. The instrumented Datalog program is first compiled into relational algebra and then into SQL.

Fig. 2: Instrumentation: **(a)** SQL, **(b)** transactions, **(c)** Datalog

P1. Blow-up in Expression Size. The instrumentation for transaction provenance [10] shown in Fig. 2b 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 results in **`SELECT`** clause expressions of size exponential in the number of stacked query blocks. While advanced optimizers do not apply this transformation unconditionally, they will at least consider it leading to the same blow-up in expression size during optimization.

P2. Common Subexpressions. Pipeline L3 [11] (Fig. 2c) instruments the input Datalog program to capture rule derivations. Compiling such queries into relational algebra leads to queries with many common subexpressions and duplicate elimination operators. Pipeline L4 constructs the PROV output using multiple projections over an instrumented subquery that captures provenance. The large number of common subexpressions in both cases may significantly increase optimization time. Furthermore, if subexpressions are not reused then this significantly increases the query size. The choice of when to remove duplicates significantly impacts performance for Datalog queries.

P3. Blocking Join Reordering. Provenance instrumentation in GProM [4] 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 query size and lead to interleaving of joins with operators such as aggregation. This interleaving may block optimizers from reordering joins leading to suboptimal join orders.

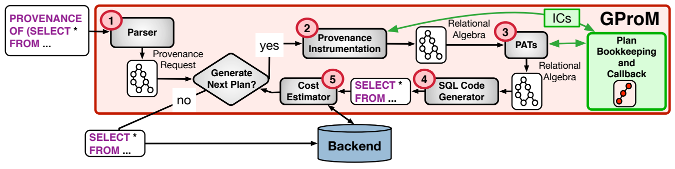


Fig. 3: GProM with Çost-based Optimizer

P4. Redundant Computations. To capture provenance, systems such as Perm [3] instrument a query one operator at a time using operator-specific rewrite rules. To apply operator-specific rules to rewrite a complex query, the rules have to be generic enough to be applicable no matter how operators are combined. This can lead to redundant computations, e.g., an instrumented operator generates a new column that is not needed by downstream operators.

2 Solution Overview

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 [14], [15], [16]. We study the orthogonal problem of **improving the performance of instrumented queries** that capture provenance. Specifically, we develop heuristic and cost-based optimization techniques to address the performance bottlenecks of instrumentation.

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. When optimizing a pipeline, we can either target one of its intermediate languages or the compilation steps. As an example for the first type of optimization, consider a compilation step that outputs relational algebra. We can optimize the generated algebra expression using algebraic equivalences before passing it on to the next stage of the pipeline. For the second type of optimization consider the compilation step from pipeline L1 that translates annotated relational algebra (with provenance) into relational algebra. If we know two equivalent ways of translating an algebra operator with annotated semantics into standard relational algebra, then we can optimize this step by choosing the translation that maximizes performance. We study both types of optimization. For the first type, we focus on relational algebra since it is an intermediate language used in all of the pipelines from Sec. 1.1. We investigate algebraic equivalences that are beneficial for instrumentation, but which are usually not applied by database optimizers. We call this type of optimizations *provenance-specific algebraic transformations* (*PATs*). We refer to optimizations of the second type as *instrumentation choices* (ICs).

PATs. We identify algebraic equivalences which 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. We infer local and non-local properties [17] 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 for provenance capture: 1) using a *join* [3] 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 [10]. Furthermore, we present two options for normalizing the output of a sequenced temporal query (L6).

Note that virtually all pipelines that we support use relational algebra as an intermediate language. Thus, our PATs are more generally applicable than the ICs which target a compilation step that only is used in some pipelines. This is however an artifact of the pipelines we have chosen. In principle, one could envision ICs that are applied to a compilation step that is common to many pipelines.

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 that 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 function 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 one execution of the pipeline as an *iteration*. It is the responsibility of the 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 an optimization choice only requires adding a few lines of code (LOC) to the pipeline to inform the optimizer about the availability of options. To the best of our knowledge our framework is the first CBO that is **plan space and query language agonistic**. Costing a plan (SQL query) requires us to use the DBMS to optimize a query which can be expensive. Thus, we may not be able to explore the full plan space. In addition to meta-heuristics, we also support a strategy that balances optimization vs. execution time.

We have implemented these optimizations in GProM [4], our provenance middleware that supports multiple DBMS backends and all the instrumentation pipelines discussed in Sec. 1.1. GProM is available as open source (https://github.com/IITDBGroup/gprom). Using L1 as an example, Fig. 3 shows how ICs, PATs, and CBO are integrated into the system. We demonstrate experimentally that our optimizations improve performance by over 4 orders of magnitude on average compared to unoptimized instrumented queries. Our approach peacefully coexists with the DBMS optimizer.

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 1: Relational algebra operators

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.

3 Background and Notation

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 schema is the number of attributes in the schema. 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 schema $\mathbf{R}$ is a function $\mathcal{U}^n \to \mathbb{N}$ mapping tuples to their multiplicity. Here $R(t)$ denotes applying the function that is R to input t, i.e., the multiplicity of tuple t in relation R. We require that relations have finite support $\text{SUPP}(R) = \{t \mid R(t) \neq 0\}$. 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 $t \in \text{SUPP}(R)$. An n-ary relation R is contained in a relation S, written as $R \subseteq S$, 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.

Table 1 shows the definition of the bag semantics version of relational algebra we use in this work. We use $\text{SCH}(Q)$ to denote the schema of the result of query Q and $Q(I)$ to denote the result of evaluating query Q over database instance I. Selection $\sigma_\theta(R)$ returns all tuples from relation R which satisfy the condition θ. Projection $\Pi_A(R)$ projects all input tuples on a list of projection expressions. Here, A denotes a list of expressions with potential renaming (denoted by $e \to a$) and $t.A$ denotes applying these expressions to a tuple t. The syntax of projection expressions is defined by the grammar shown below where **const** denotes the set of constants, **attr** denotes attributes, **c** defines conditions, and **v** defines projection expressions.

$$\begin{aligned} \mathbf{v} &:= \mathbf{v} + \mathbf{v} \mid \mathbf{v} \cdot \mathbf{v} \mid \mathbf{const} \mid \mathbf{attr} \mid \textsf{if } \mathbf{c} \textsf{ then } \mathbf{v} \textsf{ else } \mathbf{v} \\ \mathbf{c} &:= \mathbf{v}\, \mathbf{cmp}\, \mathbf{v} \mid \mathbf{c} \wedge \mathbf{c} \mid \mathbf{c} \vee \mathbf{c} \mid \neg \mathbf{c} \\ \mathbf{cmp} &:= = \mid \neq \mid < \mid \leq \mid \geq \mid > \end{aligned}$$

For instance, a valid projection expression over schema $R(a,b)$ is $(a+b) \cdot 5$. The expression type $\textsf{if } \mathbf{c} \textsf{ then } \mathbf{v} \textsf{ else } \mathbf{v}$ is introduced to support conditional expressions similar to SQL's **CASE**. The semantics of projection expressions is defined using a function $\textsf{eval}(t, e)$ which returns the result of evaluating e over t. In the following we will often use $t.e$ to denote $\textsf{eval}(t, e)$. The definition of $\textsf{eval}$ and an example for how to apply it are shown in Appendix B.1.

Union $R \cup S$ returns the bag union of tuples from relations R and S. Intersection $R \cap S$ returns the tuples which are both in relation R and S. Difference $R - S$ returns the tuples in relation R which are not in S. These set operations are only defined for inputs of the same arity. Aggregation ${}_G\gamma_{f(a)}(R)$ groups tuples according to their values in attributes G and computes the aggregation function f over the bag of values of attribute a for each group. We also allow the attribute storing $f(a)$ to be named explicitly, e.g., ${}_G\gamma_{f(a) \to x}(R)$ renames $f(a)$ as x. Duplicate removal $\delta(R)$ removes duplicates. $R \times S$ is the cross product for bags (input multiplicities are multiplied). For convenience we also define join $R \bowtie_\theta S$ and natural join $R \bowtie S$ in the usual way. For each tuple t, the window operator $\omega_{f(a) \to x, G \Vert O}(R)$ returns t with an additional attribute x storing the result of the aggregation function f. Function f is applied over the window (bag of values from attribute a) generated by partitioning the input on $G \subseteq \text{SCH}(R)$ and including only tuples which are smaller than t wrt. their values in attributes $O \subseteq \text{SCH}(R)$ where $G \cap O = \emptyset$. An example is shown in Appendix B.2. We use the window operator to express a limited form of SQL's `OVER` clause.

$$\begin{tikzcd} & \cup & \\ \Pi_a \arrow[ur] & & \Pi_b \arrow[ul] \\ & \sigma_{c<5} \arrow[ul] \arrow[ur] & \\ & R \arrow[u] & \end{tikzcd} \qquad \begin{tikzcd} & \cup & \\ \Pi_a \arrow[ur] & & \Pi_b \arrow[ul] \\ \sigma_{c<5} \arrow[u] & & \sigma_{c<5} \arrow[u] \\ R \arrow[u] & & R \arrow[u] \end{tikzcd}$$

$$\begin{aligned} Q_1 &= \Pi_a(Q_{sub}) \cup \Pi_b(Q_{sub}) \\ Q_{sub} &= \sigma_{c<5}(R) \\ Q_2 &= \Pi_a(\sigma_{c<5}(R)) \\ &\quad \cup \Pi_b(\sigma_{c<5}(R)) \end{aligned}$$

Fig. 4: Algebra graph (Q_1, left), equivalent algebra tree (Q_2, middle), and corresponding algebra expressions (right)

We represent algebra expressions as DAGs (Directed Acyclic Graph) to encode reuse of subexpressions. For instance, Figure 4 shows an algebra graph (left) which reuses an expression $\sigma_{c<5}(R)$ and the corresponding algebra tree (right). We assume that nodes are uniquely identified within such graphs and abusing notation will use operators to denote nodes in such graphs. We use $Q[Q_1 \leftarrow Q_2]$ to denote the result of substituting subexpression (subgraph) Q_1 with Q_2 in the algebra graph for query Q. Again, we assume some way of identifying subgraphs. We use $Q = op(Q')$ to denote that operator op is the root of the algebra graph for query Q and that subquery Q' is the input to op.

4 Properties and Inference Rules

We now discuss how to infer local and non-local properties of operators within the context of a query. Similar to Grust et al. [17], we use these properties in preconditions of algebraic rewrites (PATs). PATs are covered in Sec. 5.

4.1 Operator Properties

keys. Property $keys$ is a set of super keys for an operator's output. For example, if $keys(R) = \{\{a\}, \{b, c\}\}$ for a relation $R(a, b, c, d)$, then the values of attributes $\{a\}$ and $\{b, c\}$ are unique in R.

Definition 1. *Let Q be a query. A set $E \subseteq \text{SCH}(Q)$ is a* super key *for Q iff for every instance I we have $\forall t, t' \in Q(I) : t.E = t'.E \to t = t'$ and $\forall t : Q(I)(t) \leq 1$. A super key is called a candidate key if it is minimal.*

Since we are using bag semantics, in the above definition we need to enforce that a relation with a superkey cannot contain duplicates. Recall that we defined bag relations as functions, thus, $Q(I)(t)$ denotes the multiplicity of t in the result of Q over I. Klug [18] demonstrated that computing the set of functional dependencies that hold over the output of a query expressed in relational algebra is undecidable. The problem studied in [18] differs from our setting in two aspects: 1) we only consider keys and not arbitrary functional dependencies and 2) we consider a more expressive algebra over bags which includes generalized projection. As the reader might already expect, the undecidability of the problem caries over to our setting.

Theorem 1. *Computing the set of candidate keys for the output of a query Q expressed in our bag algebra is undecidable. The problem stays undecidable even if Q consists only of a single generalized projection, i.e., it is of the form $Q = \Pi_A(R)$.*

Proof. We prove the theorem by a reduction from the undecidable problem of checking whether a multi-variant polynomial over the integers ($\mathbb{Z}$) is injective. The undecidability of injectivity stems from the fact that this problem can be reduced to Hilbert's tenth problem [19] (does a Diophantine equation have a solution) which is known to be undecidable for integers. Given such a polynomial function $f(x_1, \ldots, x_n)$ over $\mathbb{Z}$, we define a schema $R(x_1, \ldots, x_n)$ over domain $\mathbb{Z}$ with a candidate key $X = \{x_1, \ldots, x_n\}$ and a query $Q_f = \Pi_{f(x_1,\ldots,x_n) \to b}(R)$. Intuitively, the query computes the set of results of f for the set of inputs stored as tuples in R. For instance, consider the multivariant polynomial $f(x, y) = x^2 + x \cdot y$. We would define an input relation R with schema $\text{Sch}(R) = (x, y)$ and query $Q_f = \Pi_{(x \cdot x + x \cdot y) \to b}(R)$ which computes f.

Now for sake of contradiction assume that we have a procedure that computes the set of candidate keys for a query based on keys given for the relations accessed by the query. The result schema of query Q_f for polynomial f consists of a single attribute (b). Thus, it has either a candiate key $\{b\}$ or no candidate key at all. Since X is a candidate key for R, $\{b\}$ is a candidate key iff f is injective (we prove this equivalence below). Thus, the hypothetical algorithm for computing the candidate keys of a query result relation gives us a decision procedure for f's injectivivity. However, deciding whether f is injective is undecidable and, thus, the problem of computing candidate keys for query results has to be undecidable.

We still need to prove our claim that $\{b\}$ is a candidate key iff f is injective.

$\Rightarrow$: For sake of contradiction assume that $\{b\}$ is a candidate key, but f is not injective. Then there have to exist two inputs $I = (i_1, \ldots, i_n)$ and $J = (j_1, \ldots, j_n)$ with $I \neq J$ such that $f(I) = y$ and $f(J) = y$ for some value y. Now consider an instance of relation R defined as $\{I, J\}$. The result of evaluating query Q_f over this instance is clearly $\{(y)^2\}$. That is, tuple (y) appears twice in the result. However, this violates the assumption that $\{b\}$ is a candidate key.

$\Leftarrow$: For sake of contradiction assume that f is injective, but $\{b\}$ is not a candidate key. Then there has to exists some instance of R such that $Q_f(R)$ contains a tuple t with multiplicity $n > 1$. Since X is a candiate key of R, we know that there are no duplicates in R. Thus, based on the definition of projection, the only way t can appear with a multiplicity larger than one is if there are two inputs t_1 and t_2 in the input such that $f(t_1) = f(t_2)$ which contradicts the assumption that f is injective. □

Given this negative result, we will focus on computing a set of keys that is not necessarily complete nor is each key in this set guaranteed to be minimal. This is unproblematic, since we will only use the existence of keys as a precondition for PATs. That is, we may miss a chance of applying a transformation since our approach may not be able to determine that a key holds, but we will never incorrectly apply a transformation.

set. Boolean property *set* denotes whether the number of duplicates in the result of a subquery Q_{sub} of a query Q is insubstantial for computing Q. We model this condition using query equivalence, i.e., if we apply duplicate elimination to the result of Q_{sub}, the resulting query is equivalent to the original query Q.

Definition 2. *Let Q_{sub} be a subquery of a query Q. We say Q_{sub} is* duplicate-insensitive *if $Q \equiv Q[Q_{sub} \leftarrow \delta(Q_{sub})]$.*

The *set* property is useful for introducing or removing duplicate elimination operators. However, as the following theorem shows, determining whether a subquery is duplicate-insensitive is undecidable. We, thus, opt for an approach that is sound, but not complete.

Theorem 2. *Let Q_{sub} be a subquery of a query Q. The problem of deciding whether Q_{sub} is duplicate-insensitive is undecidable.*

Proof. See Appendix C in the supplementary document. □

ec. The *ec* property stores a set of equivalence classes (ECs) with respect to an equivalence relation $\simeq$ over attributes and constants. Let $a, b \in (\text{Sch}(Q_{sub}) \cup \mathcal{U})$ for a subquery Q_{sub} of a query Q. We consider $a \simeq b$ if to evaluate Q we only need tuples from the result of Q_{sub} where $a = b$ holds. We model this condition using query equivalence: if $a \simeq b$ for a subquery Q_{sub} of a query Q then $Q \equiv Q[Q_{sub} \leftarrow \sigma_{a=b}(Q_{sub})]$.

Definition 3. *Let Q_{sub} be a subquery of query Q and $a, b \in (\text{Sch}(Q_{sub}) \cup \mathcal{U})$. We say a is equivalent to b, written as $a \simeq b$, if $Q \equiv Q[Q_{sub} \leftarrow \sigma_{a=b}(Q_{sub})]$. A set $E \subseteq (\text{Sch}(Q_{sub}) \cup \mathcal{U})$ is an* equivalence class *(EC) for Q_{sub} if we have $\forall a, b \in E : a \simeq b$. An EC E is maximal if no superset of E is an EC.*

As a basic sanity check we prove that $\simeq$ is in fact an equivalence relation.

Lemma 1. *$\simeq$ is an equivalence relation.*

Proof. See Appendix C in the supplementary document. □

Note that our definition of equivalence class differs from the standard definition of this concept. In fact, what is typically considered to be an equivalence class is what we call maximal equivalence class here. We consider non-maximal equivalence classes, because, as the following theorem shows, we cannot hope to find an algorithm that computes all equivalences that can be enforced for a query using generalized projection.

Rule	Operator $\diamond$	Inferred property *set* for the input(s) of $\diamond$
1,2	$\circledast$ or ${}_G\gamma_{F(a)}(R)$	$set(\circledast) = false, set(R) = false$
3,4	$\sigma_\theta(R)$ or $\Pi_A(R)$	$set(R) = set(R) \wedge set(\diamond)$
5	$\delta(R)$	$set(R) = set(R)$
6-9	$R \bowtie_{a=b} S$ or $R \times S$ or $R \cup S$ or $R \cap S$ or $R - S$	$set(R) = set(R) \wedge set(\diamond)$ $set(S) = set(S) \wedge set(\diamond)$
10	$R - S$	$set(R) = false$ $set(S) = false$
11	$\omega_{f(a)\rightarrow x,G\|O}(R)$	$set(R) = false$

TABLE 2: Top-down inference of Boolean property *set*

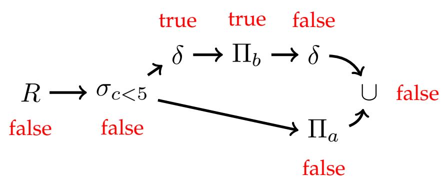

true
true
false
$\delta \longrightarrow \Pi_b \longrightarrow \delta$
$R \longrightarrow \sigma_{c<5}$
$\cup$
false
false
false
Π_a
false

Fig. 5: Inferring property *set*

Theorem 3. *Let Q_{sub} be a subquery of a query Q and $a, b \in$ $\text{SCH}(Q_{sub})$. Determining whether $a \simeq b$ is undecidable.*

Proof. See Appendix C in the supplementary document. □

In the light of this undecidability result, we develop inference rules for property ec (Section 4.2) that are sound, but not complete. That is, all inferred equivalences hold, but there is no guarantee that we infer all equivalences that hold. Put differently, the equivalence classes computed using these rules may not be maximal.

icols. This property records a set of attributes that are sufficient for evaluating the ancestors of an operator. By sufficient, we mean that if we remove other attributes this will not affect the result of the query.

Definition 4. *Let Q be a query and Q_{sub} be a subquery of Q, a set of attributes $E \subseteq \text{SCH}(Q_{sub})$ is called* sufficient *in Q_{sub} wrt. Q if $Q \equiv Q[Q_{sub} \leftarrow \Pi_E(Q_{sub})]$.*

For example, attribute d in $\Pi_a(\Pi_{a,b+c\rightarrow d}(R))$ is not needed to evaluate Π_a. Note that there exists at least one trivial set of sufficient attributes for any query Q_{sub} which is $\text{SCH}(Q_{sub})$. Ideally, we would like to find sufficient attribute sets of minimal size to be able to reduce the tuple size of intermediate results and to remove operations that generate attributes that are not needed. Unfortunately, it is undecidable to determine a minimal sufficient set of attributes.

Theorem 4. *Let Q_{sub} be a subquery of a query Q and let $E \subset$ $\text{SCH}(Q_{sub})$. The problem of determining whether E is sufficient is undecidable.*

Proof. See Appendix C in the supplementary document. □

The icols property we infer for an operator is guaranteed to be a sufficient set of attributes for the query rooted at this operator, but may not represent the smallest such set.

4.2 Property Inference

We infer properties for operators through traversals of the algebra graph of an input query. 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 initialized to a fixed value and is then updated based on the value of P for one of the parents of *op*. We use $\diamond$ to denote the operator for which we are inferring a property (for bottom-up inference) or for a parent of this operator (for top-down inference). Thus, a top-down rule $P(R) = P(R) \cup P(\diamond)$ has to be interpreted as update property P for R as the union of the current value of P for R and the current value of P for operator $\diamond$ which is a parent of R. We use $\circledast$ to denote the root of a query graph. Because of space limitations we only show the inference rules of property set here (Table 2). We show the inference rules for the remaining properties (ec, $icols$ and key) in Appendix D. In the following when to referring to properties such as the sufficient set of attributes of an operator we will implicitly understand this to refer to the property of the subquery rooted at this operator. We prove these rules to be correct in Appendix E. Here by correct we mean that $key(op)$ is a set of superkeys for op which is not necessarily complete nor does it only contain candidate keys, $ec(op)$ is a set of equivalence classes for op which may not be maximal, if the $set(op) = true$ than op is duplicate-insensitive (but not necessarily vice versa), and finally $icols(op)$ is a sufficient set of attributes for op.

Inferring the set Property. We compute set in a top-down traversal (Tab. 2). We initialize this property to true for all operators. As mentioned above our inference rules for this property are sound (if $set(op) = true$ then the operator is duplicate-insensitive), but not complete. We set $set(\circledast)$ for the root operator ($\circledast$) to false (rule 1) since the final query result will differ if duplicates are eliminated from the output of $\circledast$. Descendants of a duplicate elimination operator are duplicate-insensitive, because the duplicate elimination operator will remove any duplicates that they produce. The exception are descendants of operators such as aggregation and the window operator which may produce different result tuples if duplicates are removed. These conditions are implemented by the inference rules as follows: 1) $set(op) = true$ if op is the child of a duplicate elimination operator (Rule 5); 2) $set(op) = false$ if op is the child of a window, difference, or aggregation operator (Rules 11, 2, and 10); and otherwise 3) $set(op)$ is true if $set(\diamond)$ is true for all parents of the operator (Rules 1, 3, 4, 6-10).

Example 2. *Consider the algebra graph shown in Fig. 5. We show set for each operator as red annotations. For the root operator we set $set(\cup) = false$. Since the root operator is a union, both children of the root inherit $set(op) = false$. We set $set(\Pi_b) = true$ since Π_b is a child of a duplicate elimination operator. This propagates to the child of this projection. The selection's set property is false, because even though it is below a duplicate elimination operator, it also has a parent for which set is false. Thus, the result of the query may be affected by eliminating duplicates from the result of the selection. Finally, operator R inherits the set property from its parent which is a selection operator.*

5 PATs

We now introduce a subset of our PAT rules (Fig. 6), prove their correctness, and then discuss how these rules address the performance bottlenecks discussed in Sec. 1.2. A rule $\frac{pre}{q \rightarrow q'}$ has to be read as "If condition pre holds, then q can be

$$\frac{a \subseteq \textsc{Sch}(\Diamond(\Pi_A(R)))}{\Diamond(\Pi_{A,a\to b}(R)) \to \Pi_{\textsc{Sch}(\Diamond(\Pi_A(R))),a\to b}(\Diamond(\Pi_A(R)))} \quad (1) \qquad \frac{keys(R) \neq \emptyset}{\delta(R) \to R} \quad (2) \qquad \frac{set(\delta(R))}{\delta(R) \to R} \quad (3) \qquad \frac{A = icols(R)}{R \to \Pi_A(R)} \quad (4)$$

$$\frac{G \subseteq \textsc{Sch}(R)}{_G\gamma(R \bowtie_{b=c} S) \to {_G\gamma}(_{G,b}\gamma(R) \bowtie_{b=c} S)} \quad (5) \qquad \frac{e_1 = \text{if } \theta \text{ then } a + c \text{ else } a}{\Pi_{e_1,\dots,e_m}(R) \to \Pi_{a+\text{if } \theta \text{ then } c \text{ else } 0, e_2,\dots,e_m}(R)} \quad (6)$$

$$\frac{x \notin icols(\omega_{f(a)\to x, G\|O}(R))}{\omega_{f(a)\to x,G\|O}(R) \to R} \quad (7) \qquad \frac{a \in \textsc{Sch}(R) \land a \notin (G \cup \{b,c\}) \land b \in G \land G \subseteq \textsc{Sch}(R) \land \{c\} \in keys(S)}{_G\gamma_{f(a)}(R \bowtie_{b=c} S) \to {_G\gamma_{f(a)}}(R) \bowtie_{b=c} S} \quad (8)$$

Fig. 6: Provenance-specific transformation (PAT) rules

rewritten as q'''. Note that we also implement standard optimization rules such as selection move-around, and merging of adjacent projections, because these rules may help us to fulfill the preconditions of PATs (see Appendix F).

Provenance Projection Pull Up. Provenance instrumentation [4], [3] seeds provenance annotations by duplicating attributes of input relations 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 $a \to b$ through operator $\Diamond$. For example, 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: $\Pi_{a,b,a\to P(a),b\to P(b)}(\sigma_{a<5}(R))$.

Remove Duplicate Elimination. Rules (2) and (3) remove duplicate elimination operators. If a relation R has at least one super 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 duplicates (property *set* is true for *op*), then *op* can be removed (Rule (3)).

Remove Redundant Attributes. Recall that $icols(R)$ is a set of attributes from relation R which is sufficient to evaluate ancestors of R. If $icols(R) = A$, then we use Rule (4) to remove all other attributes by projecting R on A. Operator $\omega_{f(a)\to x,G\|O}(R)$ extends each tuple $t \in R$ by adding a new attribute x that stores the aggregation function result $f(a)$. Rule (7) removes ω if x is not needed by ancestors of $\omega_{f(a)\to x,G\|O}(R)$.

Attribute Factoring. Attribute factoring restructures projection expressions such that adjacent projections can be merged without blow-up in expression size. For instance, merging projections $\Pi_{b+b+b\to c}(\Pi_{a+a+a\to b}(R))$ 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. Recall that we represent **CASE** as if θ then e_1 else e_2 in projection expressions. For example, update `UPDATE R SET a = a + 2 WHERE b = 2` would be expressed as $\Pi_{\text{if } b=2 \text{ then } a+2 \text{ else } a, b}(R)$ which can be rewritten as $\Pi_{a+\text{if } b=2 \text{ then } 2 \text{ else } 0, b}(R)$, reducing the references to a by 1. We define analog rules for any arithmetic operation which has a neutral element (e.g., multiplication).

Aggregation Push Down. Pipeline L5 encodes the provenance (provenance polynomial) of a query result as an XML document. Each polynomial is factorized based on the structure of the query. We can reduce the output's size by rewriting the query using algebraic equivalences to choose a beneficial factorization [20]. For example, $a{\cdot}b{+}a{\cdot}c{+}a{\cdot}d$ can be factorized as $a \cdot (b + c + d)$. For queries with aggregation, this factorization can be realized by pushing aggregations through joins. Rule (5) and (8) push down aggregations based on the equivalences introduced in [21]. Rule 8 pushes an aggregation to a child of a join operator if the join is cardinality-preserving and all attributes needed to compute the aggregation are available in that child. For instance, consider $_b\gamma_{f(a)}(R \bowtie_{b=c} S)$ where $\{c\}$ is a key of S. Since R is joined with S on $b = c$, pushing down the aggregation to R does not affect the cardinality of the aggregation's input. Since also $\{a, b\} \in \textsc{Sch}(R)$, we can rewrite this query into $_b\gamma_{f(a)}(R) \bowtie_{b=c} S$. Rule 5 redundantly pushes an aggregation without aggregation functions (equivalent to a duplicate elimination) to create a pre-aggregation step.

Theorem 5. *The PATs from Fig. 6 are equivalence preserving.*

Proof. See Appendix F in the supplementary document. □

5.1 Addressing Bottlenecks through PATs

Rule (6) is a preprocessing step that helps us to avoid a blow-up in expression size when merging projections (Sec. 1.2 **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 (7) remove such operators. Even if such operators cannot be removed, Rules (1) and (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 (7) to remove redundant operators. Furthermore, Rule (4) removes unnecessary columns. Rule (5) and (8) factorize nested representations of provenance (Pipeline L5) to reduce its size by pushing aggregations through joins. 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 *ec* property), merge 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). These additional PATs are discussed in Appendix F.

6 Instrumentation Choices

Window vs. Join. The *Join* method for instrumenting an aggregation operator for provenance capture was first used by Perm [3]. 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 aggregation's 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 [3] for details). For instance, consider a query ${}_{b}\gamma_{sum(a)\rightarrow x}(R)$ with $\textsc{Sch}(R) = (a, b)$. This query would be rewritten into $\Pi_{b,x,P(a),P(b)}({}_{G}\gamma_{sum(a)\rightarrow x}(R) \bowtie_{b=b'} \Pi_{b\rightarrow b',a\rightarrow P(a),b\rightarrow P(b)}(R))$. 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 $\Pi_{b,x,P(a),P(b)}(\omega_{sum(a)\rightarrow x,b\|}(R)(\Pi_{a,b,a\rightarrow P(a),b\rightarrow P(b)}(R)))$. The *Window* method has the advantage that no additional joins are introduced. However, as we will show in Sec. 9, 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 [10] only returns the provenance of tuples that were affected by T. We consider 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 [10]. *FilterUpdated* is typically superior, because it avoids the join applied by *HistJoin*. However, for transactions with a large number of operations, the cost of *FilterUpdated*'s selection can be higher than the join's cost.

Set-coalesce vs. Bag-coalesce. The result of a sequenced temporal query [13] can be encoded in multiple, equivalent ways using intervals. Pipeline L6 applies a normalization step to ensure a unique encoding of the output. Coalescing [22], the standard method for normalizing interval representations of temporal data under set semantics, is not applicable for bag semantics. We introduce a version that also works for bags. However, this comes at the cost of additional overhead. If we know that a query's output does not contain any duplicates, then we can use the cheaper set-coalescing method. We use Property key to determine whether should we can apply *set-coalesce* (see Appendix A.2).

7 Cost-based Optimization

Our CBO algorithm (Alg. 1) 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 the current plan P is less than the cost T_{best} of the best plan found so far, then we set $P_{best} = P$. Finally, we decide which optimization choices to make in the next iteration using function genNextIterChoices. Our optimizer is plan space agnostic. New choices are discovered at runtime when a step in the pipeline informs the optimizer about an optimization choice. This enables the optimizer to enumerate all plans for a blackbox instrumentation pipeline.

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:              $T_{best} \leftarrow T, P_{best} \leftarrow P$
9:          genNextIterChoices( )
10:         $T_{opt} = T_{opt} + ($currentTime() $- t_{before})$
11:     return $P_{best}$
```

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 cost based on the properties of the query alone.

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).

7.1 Registering Optimization Choices

We want to make the optimizer aware of choices available in a pipeline without having to significantly change existing code. Choices are registered by calling the optimizer's makeChoice function. This callback interface has two purposes: 1) inform the optimizer that a choice has to be made and how many alternatives to choose from and 2) allowing it to control which options are chosen. 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 return value of makeChoice instructs the caller to take a particular option.

Example 3. *Assume we want to make a cost-based decision on whether to use the* Join *or* Window *method (Sec. 6) to instrument an aggregation. We add a call* makeChoice(2) *to register a choice with two options to choose from. The optimizer responds with a number (0 or 1) encoding the option to be chosen.*

```
if (makeChoice(2) == 0) Window(Q) else Join(Q)
```

A code fragment containing a call to makeChoice may be executed several times during one iteration. Every call is treated as an independent choice point, e.g., 4 possible combinations of the *Join* and *Window* methods will be considered for instrumenting a query with two aggregations.

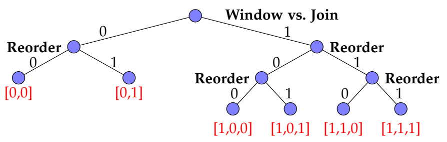


Fig. 7: Plan space tree example

7.2 Plan Enumeration

During one iteration we may hit any number of choice points and each choice made may affect 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 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 4. *Assume we use 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. Consider a query which is an aggregation over the result of a join. Fig. 7 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 have to decide wether to reorder the inputs of the join. 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.*

While the plan space tree encodes all possible plans for a given query and set of choice points, it would not be feasible to 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 which path was taken in the previous iteration (represented as a list of choices as shown in Fig. 7) 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. We call this traversal strategy *sequential-leaf-traversal*. We have implemented an alternative strategy that approximates a binary search over the leaf nodes. We opt for an approximation, because the structure of subtrees is not known upfront. This strategy called *binary-search-traversal* is described in more detail in Appendix G.3. The rationale for supporting this strategy is that if time constraints prevent us from exploring the full search space, then we would like to increase the diversity of explored plans by traversing different sections of the plan tree.

Theorem 6. *Let Q be input query. Algorithm 1 iterates over all plans that can be created for the given choice points.*

7.3 Alternative Search Strategies

Metaheuristics are applied in query optimization to deal with large search spaces. We discuss an implementation of a metaheuristic in our framework in Appendix G.

Balancing Optimization vs. Runtime. The strategies discussed so far do not adapt the effort spend on optimization based on how expensive the query is. Obviously, spending more time on optimization than on execution is undesirable (assuming that provenance requests are ad hoc). Ideally, we would like to minimize the sum of the optimization time (T_{opt}) and execution time of the best plan T_{best} by stopping optimization once a cheap enough plan has been found. This is an online problem, i.e., after each iteration we have to decide whether to execute the current best plan or continue to produce more plans. The following stopping condition results in a 2-competitive algorithm, i.e., $T_{opt} + T_{best}$ is less than 2 times the minimal achievable cost: stop optimization once $T_{best} = T_{opt}$. Note that even though we do not know the length of an iteration upfront, we can still ensure $T_{best} = T_{opt}$ by stopping mid iteration.

Theorem 7. *The algorithm outlined above is 2-competitive.*

Proof. See Appendix G in the supplementary document. □

8 Related Work

Our work is related to optimizations that sit on top of standard CBO, to compilation of non-relational languages into SQL, and to provenance capture and storage optimization.

Cost-based Query Transformation. State-of-the-art DBMS apply transformations such as decorrelation of nested subqueries [23] in addition to (typically exhaustive) join enumeration and choice of physical operators. Often such transformations are integrated with CBO [24] 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., [25]. 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. Furthermore, our framework has the advantage that new optimization choices can be added without modifying the optimizer.

Compilation of Non-relational Languages into SQL. Approaches that compile non-relational languages (e.g., XQuery [17], [26]) or extensions of relational languages (e.g., temporal [27] and nested collection models [28]) into SQL face similar challenges as we do. Grust et al. [17] optimize the 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 their idea of inferring properties over algebra graphs. However, to the best of our knowledge we are the first to integrate these ideas with CBO and to consider ICs.

Provenance Instrumentation. Several systems such as *DBNotes* [5], *Trio* [29], *Perm* [3], *LogicBox* [2], *ExSPAN* [7], and *GProM* [4] 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 Capture and Storage. Optimization of provenance has mostly focused on minimizing the storage size of provenance. Chapman et al. [15] 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 [14]. A cost-based framework for choosing between reference-based provenance storage and propagating full provenance was introduced in the context of declarative networking [7]. 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 [4], [10] and in the Subzero system [16]. Subzero switches between different provenance storage representations in an adaptive manner to optimize the cost of provenance queries. Amsterdamer et al. [30] demonstrate how to rewrite a query into an equivalent query with provenance of minimal size. Our work is orthogonal in that we focus on minimizing execution time of provenance capture and retrieval.

9 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 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 3 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. 5 turned out to be always beneficial in our experiments. Thus, these PATs are applied as long as their precondition is fulfilled. We consider two variants for each method: activating heuristic application of the remaining PATs (suffix **Heu**) or deactivating them (**NoHeu**). Unless noted otherwise, results were averaged over 100 runs.

9.1 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 values. We vary the size of the transactional history. 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/). MESD: The temporal MySQL employees sample dataset has 6 tables and contains 4M records (https://dev.mysql.com/doc/employee/en/).

Simple aggregation queries. This workload computes the provenance of queries consisting solely of aggregations using Pipeline L1 which applies the rewrite rules for aggregation pioneered in Perm [3] and extended in GProM [4]. A query consists of i aggregations 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 PK values such that the last step returns the same number of results independent of i.

TPC-H queries. We select 11 out of the 22 TPC-H queries to evaluate optimization of provenance capture for complex queries. The technique [31] we are using supports all TPC-H queries, but instrumentations for nested subqueries have not been implemented in GProM yet.

Transactions. We use the *reenactment* approach of GProM [10] to compute provenance for transactions executed under isolation level `SERIALIZABLE`. 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 PK of the relation.

Provenance export. We use the approach from [12] to translate a relational encoding of provenance (see Sec. 1) into PROV-JSON. We export the provenance for a foreign key join across TPC-H relations nation, customer, and orders.

Provenance for Datalog queries. We use the approach described in [11] (Pipeline L3). The input is a non-recursive Datalog query Q and a set of (missing) query result tuples of interest. We use the DBLP co-author dataset for this experiment and the following queries. **Q1**: Return authors which have co-authors that have co-authors. **Q2**: Return authors that are co-authors, but not of themselves (while semantically meaningless, this query is useful for testing 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.

Factorizing Provenance. We use Pipelines L1 and L5 to evaluate the performance of nested versus "flat" provenance under different factorizations (applying the aggregation push-down PATs). We use the following queries over the TPC-H dataset. **Q1**: An aggregation over a join of tables customer and nation. **Q2**: Joins the result of Q1 with the table supplier and adds an additional aggregation. **Q3**: An aggregation over a join of tables nation, customer, and supplier.

Sequenced temporal queries. We use Pipeline L6 to test the IC which replaces *bag-coalesce* with *set-coalesce* for queries that do not return duplicates. We use the following queries over the temporal MESD dataset. **Q1**: Return the average salary of employees per department. **Q2**: Return the salary and department for every employee (3-way join).

9.2 Measuring Query Runtime

Overview. Fig. 17 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 workloads are significantly more expensive than other. 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 reported 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% worse than 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. Using a suboptimal heuristic on average increases runtime by a factor of ~ 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.

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. 8 (the best method is shown in bold). We also show the average runtime per query relative to the runtime of *Join+NoHeu*. CBO 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. 9 and 10 show the results for individual queries. Note that the y-axis is log-scale. Activating *Heu* improves performance in most cases, but the dominating factor for this workload is choosing the right method for instrumenting aggregations. The exception is the *Join* method, where runtime increases when *Heu* 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 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, CBO did either select *Window* as the superior method (confirmed by inspecting the generated execution plan) or did outperform both *Window* and *Join* by instrumenting some aggregations 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. 8. We also show the average runtime per query relative to the runtime of *Join+NoHeu*. Fig. 11 and 12 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 *Join+Heu*. Note that the runtime of *Join+Heu* 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 aggregations, all other queries only contain one aggregation. The CBO was able to determine the best method to use in almost all cases. Inferior choices are again caused by inaccurate cost estimates. We also show the results for *NoHeu*. However, only three queries finished within the allocated time slot of 6 hours (Q1, Q6 and Q13). These results demonstrate the need for PATs and the robustness of our CBO method.

Transactions. We next compute the provenance of transactions executed over the synthetic dataset using the techniques introduced in [10]. 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. 18. The left graph in Fig. 15 shows detailed results. We compare the runtime of *FilterUpdated* and *HistJoin* (*Heu* and *NoHeu*) with *Cost+Heu*. Our CBO choses *FilterUpdated*, the superior option.

Provenance Export. Fig. 13 shows results for the provenance export workload for dataset sizes from 10MB up to 10GB (total workload runtime is shown in Fig. 19). *Cost+Heu* and *Heu* both outperform *NoHeu* demonstrating the key role of PATs for this workload. Our provenance instrumentations use window operators for enumerating intermediate result tuples which prevents the database from pushing selections and reordering joins. *Heu* outperforms *NoHeu*, because it removes some of these window operators (PAT rule (7)). CBO does not further improve performance, because the export query does not apply aggregation or duplicate elimination, i.e., none of the choice points were hit.

Why Questions for Datalog. The approach [11] 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. 6). 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

Queries	Join+NoHeu	Join+Hue	Window+NoHeu	Window+Heu	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	Join+NoHeu	Join+Heu	Window+NoHeu	Window+Heu	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. 8: Total (**Left**) and average runtime per query (**Right**) relative to Join+NoHeu for *SimpleAgg* and *TPC-H* workloads

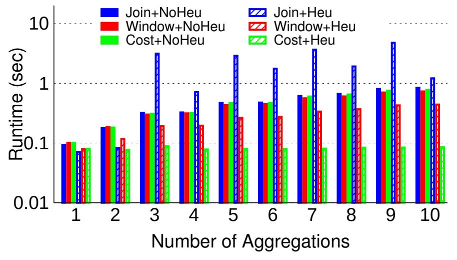


Fig. 9: 1GB *SimpleAgg* runtime

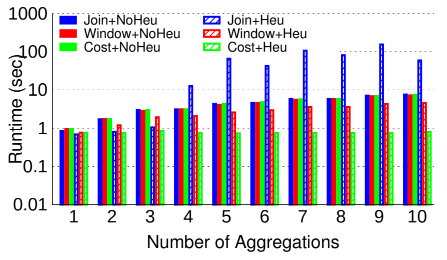


Fig. 10: 10GB *SimpleAgg* runtime

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

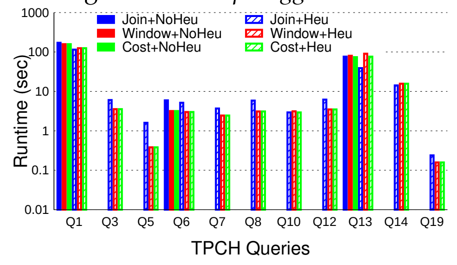


Fig. 11: Runtime *TPC-H* - 1GB

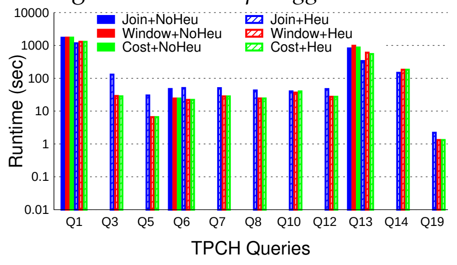


Fig. 12: Runtime *TPC-H* - 10GB

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. 18: 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. 19: Total runtime for export and Datalog workloads

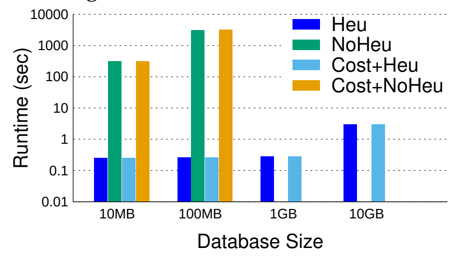


Fig. 13: Provenance Export

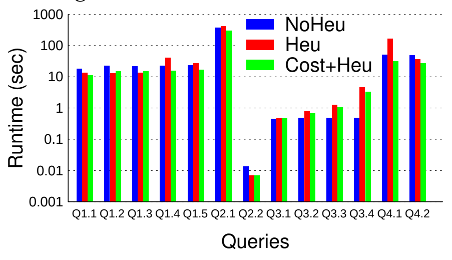


Fig. 14: Datalog Provenance

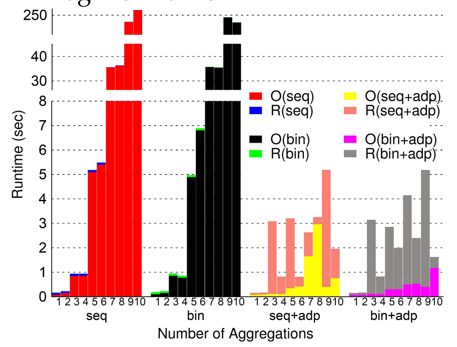


Fig. 20: Optimization + runtime for Simple Agg. - 1GB

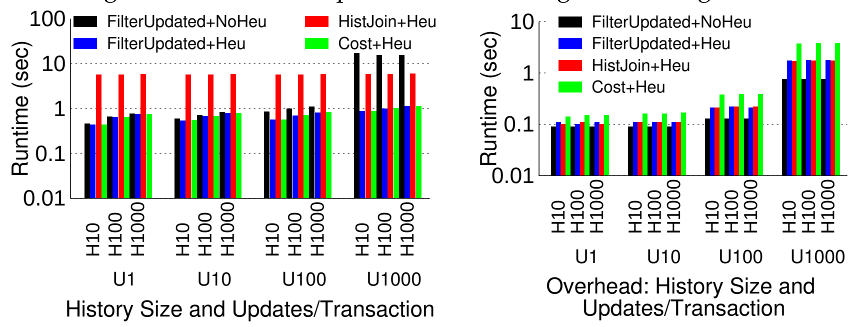


Fig. 15: Transaction provenance - runtime and overhead

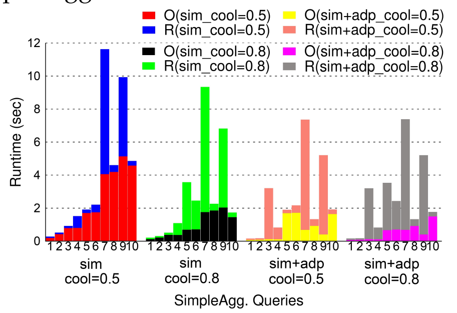


Fig. 21: Optimization + runtime for Simple Aggregation workload using Simulated Annealing - 1GB dataset

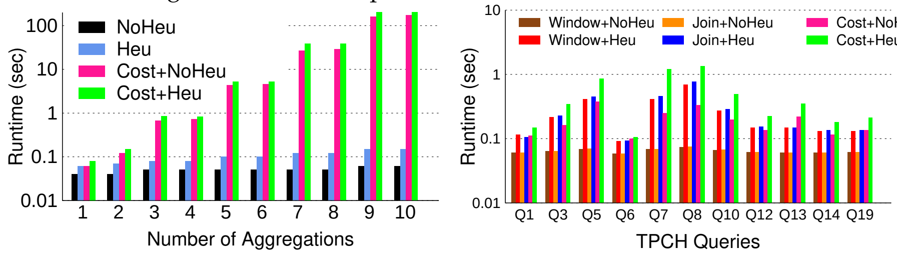


Fig. 16: *SimpleAgg* (**Left**) and *TPC-H* (**Right**) Overhead

		Normal (NoHeu)		Fac (Heu)	
Query	**Size**	**Prov**	**Xml**	**Prov**	**Xml**
Q1	10MB	0.0991	0.0538	0.1062	**0.0410**
	100MB	0.8481	0.37629	0.9302	**0.2582**
	1GB	8.4233	6.9261	8.9676	**4.8069**
	10GB	122.5400	95.1900	150.2800	**77.2800**
Q2	10MB	0.1876	0.0584	0.1386	**0.0352**
	100MB	17.5624	error	12.4271	**0.2627**
	1GB	+3600.0000	error	1329.0000	**5.3133**
	10GB	+3600.0000	error	+3600.0000	**86.9500**
Q3	10MB	0.4312	0.1357	0.4414	**0.0918**
	100MB	42.8268	35.9454	47.4869	**5.2949**

Fig. 22: Runtime for Factorization Queries (Q1 to Q3)

as an optimization choice in our CBO. The total workload runtime and results for individual queries are shown in Fig. 19 and Fig. 14, respectively. 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.

Factorizing Provenance. We compare the runtime of Pipeline L1 (**Prov**) against P5 (**XML**) which produces a nested representation of provenance. We test the effect of the heuristic application of aggregation push-down (Rules 5 and 8 from Fig. 6) to factorize provenance. Fig. 22 shows the runtimes for the factorization workload (queries Q1 to Q3). In general, *XML* outperforms *Prov* since it reduces the number of query results (rows) and total size of the results in bytes. *Prov* does not benefit much from aggregation push-down, because this does not affect the size of the returned provenance. This optimization improves performance for *XML*, specifically for larger database instances. In summary, *XML+Heu* is the fasted method in all cases, outperforming *Prov+Heu* by a factor of up to 250. Note that DBMS X does not support large XML values in certain query contexts that require sorting. A query that encounters such a situation will fail with an error message (marked in red in Fig. 22).

Set vs. Bag Coalescing. We also run sequenced temporal queries comparing *Heu* (use set-coalesce) and *NoHeu*. The result set of Query Q1 is small. Thus, using set-coalesce (*Heu*) only improves performance by ~10%. The runtimes are 4.85s *(Heu)* and 5.27s *(NoHeu)*. Choosing the right coalescing operator is more important for Query Q2 which returns 2.8M tuples (35.38s for *Heu* and 64s for *NoHeu*).

9.3 Optimization Time and CBO Strategies

Simple Aggregation. We show the optimization time of several methods in Fig. 16 (left). Heuristic optimization (*Heu*) results in an overhead of ~50ms compared to the time of compiling a provenance request without optimization (*NoHeu*). This overhead is only slightly affected by the number of aggregations. The overhead is higher for *Cost* because we have 2 choices for each aggregation, i.e., the plan space size is 2^i for i aggregations. We have measured where time is spend during CBO 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 for some queries.

	cooling rate (cr)								
Method	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
Sim	28.9	14.6	8.6	6.5	4.6	3.6	2.6	2.0	1.5
Sim+Adp	1.64	1.64	1.64	1.64	1.64	1.64	1.63	1.62	1.5

TABLE 3: Parameter Sensitivity for Simulated Annealing

TPC-H Queries. In Fig. 16 (right), we show the optimization time for TPC-H queries. Activating PATs results in ~50ms overhead in most cases with a maximum overhead of ~0.5s. This is more than offset by the gain in query performance (recall that with *NoHeu* only 3 queries finish within 6 hours 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. 7. 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. 7.3. 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. 20 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. 21 shows the total time of *sim* with and without the *adp* strategy for the same workload. We used cooling rates (*cr*) of 0.5 and 0.8 because they resulted in the best performance. The *adp* strategy improves the runtime in all cases except for the query with 3 aggregations. We also evaluated the effect of the *cr* and *c* parameters for simulated annealing ***(sim)*** and its adaptive version ***(sim+adp)*** by varying the *cr* (0.1 ~ 0.9) and *c* value (1, 100 and 10000) for Simple Aggregation query Q10 over the 1GB dataset. The choice of parameter *c* had negledible impact. Thus, we focus on *cr*. Tab. 3 shows the optimization time for *c*=10000 for these two methods. The query execution time was 0.27s for *cr* (0.1 ~ 0.8) and 1.2s for *cr* 0.9. The total cost is minimized (2.0+0.27=2.27s) when for *cr* 0.8. *sim+adp* further reduces the optimization time to roughly 1.64s independent of the *cr*.

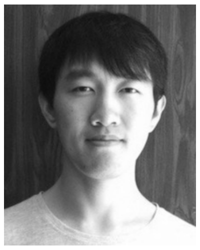
10 Conclusions and Future Work

We present the first cost-based optimization framework for provenance instrumentation and its implementation in GProM. Our approach supports both heuristic and cost-based choices and is applicable to a wide range of instrumentation pipelines. We study provenance-specific algebraic transformations (PATs) and instrumentation choices (ICs), i.e., alternative ways of realizing provenance capture. We demonstrate experimentally that our optimizations improve performance by several orders of magnitude for diverse provenance tasks. An interesting avenue for future work is to incorporate CBO with provenance compression.

Acknowledgements. This work was supported by NSF Award #1640864. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

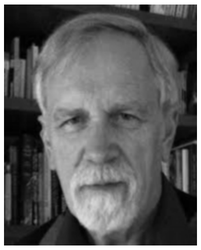
[1] G. Karvounarakis and T. Green, "Semiring-annotated data: Queries and provenance," *SIGMOD*, vol. 41, no. 3, pp. 5–14, 2012.
[2] T. J. Green, M. Aref, and G. Karvounarakis, "Logicblox, platform and language: A tutorial," in *Datalog in Academia and Industry*. Springer, 2012, pp. 1–8.
[3] B. Glavic, 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*. Springer, 2013, pp. 291–320.
[4] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng, "GProM - A swiss army knife for your provenance needs," *Data Eng. Bull.*, vol. 41, no. 1, pp. 51–62, 2018.
[5] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "An annotation management system for relational databases," *VLDBJ*, vol. 14, no. 4, pp. 373–396, 2005.
[6] S. Köhler, B. Ludäscher, and Y. Smaragdakis, "Declarative datalog debugging for mere mortals," *Datalog in Academia and Industry*, pp. 111–122, 2012.
[7] 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*, 2010, pp. 615–626.
[8] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan, "Provenance-aware query optimization," in *ICDE*, 2017, pp. 473–484.
[9] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in Databases: Why, How, and Where," *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009.
[10] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic, "Reenactment for read-committed snapshot isolation," in *CIKM*, 2016, pp. 841–850.
[11] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic, "A sql-middleware unifying why and why-not provenance," in *ICDE*, 2017.
[12] 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.
[13] M. H. Böhlen and C. S. Jensen, "Sequenced semantics," in *Encyclopedia of Database Systems*, 2009, pp. 2619–2621.
[14] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher, "Efficient provenance storage over nested data collections," in *EDBT*, 2009, pp. 958–969.
[15] A. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," in *SIGMOD*, 2008, pp. 993–1006.
[16] E. Wu, S. Madden, and M. Stonebraker, "Subzero: a fine-grained lineage system for scientific databases," in *ICDE*, 2013, pp. 865–876.
[17] T. Grust, M. Mayr, and J. Rittinger, "Let SQL drive the XQuery workhorse (XQuery join graph isolation)," in *EDBT*, 2010, pp. 147–158.
[18] A. Klug, "Calculating constraints on relational expression," *TODS*, vol. 5, no. 3, pp. 260–290, 1980.
[19] Y. V. Matiyasevich and J. E. Fenstad, *Hilbert's tenth problem*. MIT press Cambridge, 1993, vol. 105.
[20] D. Olteanu and J. Závodný, "On factorisation of provenance polynomials," in *TaPP*, 2011.
[21] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *VLDB*, 1994, pp. 354–366.
[22] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *VDLB*, 1996, pp. 180–191.
[23] P. Seshadri, H. Pirahesh, and T. Leung, "Complex Query Decorrelation," *ICDE*, pp. 450–458, 1996.
[24] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes, "Cost-based query transformation in Oracle," in *PVLDB*, 2006, pp. 1026–1036.
[25] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993, pp. 209–218.
[26] Z. H. Liu, M. Krishnaprasad, and V. Arora, "Native XQuery processing in Oracle XMLDB," in *SIGMOD*, 2005, pp. 828–833.
[27] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass, "Adaptable query optimization and evaluation in temporal middleware," in *SIGMOD*, 2001, pp. 127–138.
[28] J. Cheney, S. Lindley, and P. Wadler, "Query shredding: efficient relational evaluation of queries over nested multisets," in *SIGMOD*, 2014, pp. 1027–1038.
[29] C. C. Aggarwal, "Trio a system for data uncertainty and lineage," in *Managing and Mining Uncertain Data*. Springer, 2009, pp. 1–35.
[30] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen, "On provenance minimization," *TODS*, vol. 37, no. 4, p. 30, 2012.
[31] B. Glavic and G. Alonso, "Provenance for nested subqueries," in *EDBT*, 2009, pp. 982–993.



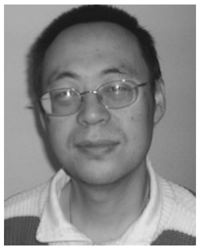
Xing Niu received a Master in CS from Henan University. He is now a PhD student at Illinois Institute of Technology.

Raghav Kapoor received a Master in CS from Illinois Institute of Technology. He is now a software engineer at Teradata.

Boris Glavic received the PhD in computer science from the University of Zurich. Currently, he is an Assistant Professor at the Illinois Institute of Technology focusing on data provenance and integration.



Dieter Gawlick is an architect at Oracle. He has developed key concepts for high-end OLTP, storage management, messaging, workflow, and information dissemination.



Zhen Hua Liu He is the architectural brain behind semi-structured data management at Oracle.

Vasudha Krishnaswamy received the PhD in computer science from the University of California, Santa Barbara working on semantics based concurrency control. Currently, she is a Consulting Member of Technical Staff at Oracle.

Venkatesh Radhakrishnan received the PhD in computer science from SUNY Albany. Currently, he is a Software Engineer at Yugabyte, working on a transactional, high-performance database for planet-scale cloud applications. Prior to that, he was a Staff Software Engineer at LinkedIn, working on Pinot, a realtime distributed OLAP datastore.