Provenance-aware Query Optimization

Xing Niu, Raghav Kapoor, Boris Glavic
Illinois Institute of Technology
{xniu7,rkapoor7}@hawk.iit.edu, bglavic@iit.edu

Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy
Oracle
{dieter.gawlick,zhen.liu,vasudha.krishnaswamy}@oracle.com

Venkatesh Radhakrishnan
LinkedIn
vradhakrishnan@linkedin.com

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 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 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 [6], DBNotes [17], LogicBox [14], declarative Datalog debugging [18], ExSPAN [20], 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 proveance 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 [10]. The example below introduces a relational encoding of provenance polynomials [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:

\[
\Pi_{\text{name}}(\text{shop} \bowtie \text{name} = \text{shop} \bowtie \text{sale} \bowtie \text{price}>20(\text{item}))
\]

The result of this query is shown in Fig. 2. Using provenance polynomials to represent provenance, tuples in the database are annotated with variables representing tuple identifiers. We show how these annotations are annotated for 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 $s_3$, $a_3$, and $i_3$ ($s_3 \cdot a_3 \cdot i_3$). Fig. 2a shows a relational encoding of these annotations as supported by the Perm [12] and GProM [6] 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. This encoding is computed by compiling the input query with annotated semantics into relational algebra. The resulting

\[
(\text{Provenance Request}) \xrightarrow{\text{Annotated Relational Algebra}} \text{Relational Algebra} \xrightarrow{\text{SQL Code Generation}} \text{SQL Code}
\]

(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.

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

(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.
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.  
\[
Q_{\text{join}} = \text{shop} \bowtie_{\text{name}=\text{shop}} \text{sale} \bowtie_{\text{id} \cdot \text{price} \geq \text{price}(\text{item})} \text{id} \cdot \text{price} \\
Q = \Pi_{\text{name} \rightarrow \text{P(name)}, \text{numEmp} \rightarrow \text{P(numEmp)}, \ldots}(Q_{\text{join}})
\]
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).

A. Instrumentation Pipelines

Provenance for SQL Queries. The instrumentation technique shown in Fig. \([13]\) and explained in the example above is applied by many relational provenance systems. For instance, the DBNotes \([1]\) system uses instrumentation to propagate attribute-level annotations according to Where-provenance \([9]\). 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 \([5]\) 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 \(\text{UPDATE R SET b = b + 2 WHERE a = 1}\) over relation \(R(a, b)\) can be reenacted using a query \(\text{SELECT a, CASE WHEN a=1 THEN b+2 ELSE b END AS b FROM R}\). 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 a tuple is or is not in the result of a Datalog query \([19]\). Such graphs record which successful and failed rule derivations 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. \([13]\) 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 \([22]\) 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 \([19], [22], [6], [5], [12]\) 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.

P1. Blow-up in Expression Size. The instrumentation for transaction provenance \([5]\) 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 \(\text{SELECT}\) 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 \([6]\) 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. To capture provenance, most provenance approaches instrument a query 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 \[4\]. 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 capture 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 \[6\] (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 PATs 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 not usually 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 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 an 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 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 agnostic. 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 \(D\) = \(\{R_1, \ldots, R_n\}\) is a set of relation schemas \(R\) \(\langle a_1, \ldots, a_n\rangle\) consists of a name (\(R\)) 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 \( \mathcal{R} \) is a function \( \mathcal{U}^n \rightarrow \mathcal{N} \) with finite support \(|\{t | R(t) \neq 0\}|\). We use \( t^n \in R \) 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 \( \gamma f(a)(R) \) 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 \( \omega f(a) \rightarrow t.x (G) (R) \) 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 \( op_1 \sim op_2 \) to denote that operator \( op_2 \) is an ancestor of (downstream of) \( op_1 \) and \( op_1 \sim op_2 \) to denote that \( op_2 \) lies on all paths from \( op_1 \) the root of the query these operators belong to. Furthermore, we use \( Q(Q_I \leftarrow Q_2) \) to denote the substitution of subexpression \( Q_1 \) in \( Q \) with \( Q_2 \) and \( \text{SCH}(Q) \) to denote the schema of the result of an algebra expression \( Q \).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>Inferred Property for the Output of ⊙</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ</td>
<td>( \sigma_R(\mathcal{I}) = { t \mid t \in R \land q } )</td>
<td>( \sigma_{R_{op_2} \rightarrow \mathcal{I}}(Q) )</td>
</tr>
<tr>
<td>( \cup )</td>
<td>( R \cup S = { t \mid t \in R \lor t \in S } )</td>
<td>( \sigma_{R_{op_2} \rightarrow \mathcal{I}}(Q) )</td>
</tr>
<tr>
<td>( \cap )</td>
<td>( R \cap S = { t \mid t \in R \land t \in S } )</td>
<td>( \sigma_{R_{op_2} \rightarrow \mathcal{I}}(Q) )</td>
</tr>
<tr>
<td>( \times )</td>
<td>( R \times S = { (t, s) \mid t \in R, s \in S } )</td>
<td>( \sigma_{R_{op_2} \rightarrow \mathcal{I}}(Q) )</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>( \gamma f(a)(R) = { (G, s) \mid s \in R } )</td>
<td>( \sigma_{R_{op_2} \rightarrow \mathcal{I}}(Q) )</td>
</tr>
<tr>
<td>( \delta )</td>
<td>( \delta R = { t</td>
<td>t \in R } )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( \omega f(a) (G) (R) = { (t, f(P(t)))</td>
<td>t \in R } )</td>
</tr>
</tbody>
</table>

Table I: Relational algebra operators

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 \( Q \). \( set(op) = \{ \text{true if } \exists op' : op \sim op' \} \) with \( op' = \delta \) and \( \forall \)op'' : op'' \sim op' we have op'' \( \notin \{ \gamma, \omega \} \).

**keys.** Keys is a set of 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 \( Q = \text{op}(Q') \) be a query. A set \( E \subseteq \text{SCH}(Q) \) 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 \( t^n \in Q(I) \rightarrow n \leq 1 \).

**EC.** The equivalence class \( (EC) \) property records for which attributes in the output of an operator we can enforce an equality condition without changing the semantic of the query. For example, if \( EC(R) = \{ \{a\}, \{b, c\}, \{d\} \} \) then we can safely enforce \( b = c \) using a selection.

**Definition 3.** Let \( Q_{\text{sub}} = \text{op}(Q'_{\text{sub}}) \) be a subquery of query \( Q \). A set of attributes \( E \subseteq \text{SCH}(Q) \) is an equivalence class \( (EC) \) for \( op \) iff for all instances \( I \) we have \( \forall t \in Q(I) : t.E = t'.E \rightarrow t = t' \) and \( t^n \in Q(I) \rightarrow n \leq 1 \).

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

**Definition 4.** Let \( Q \) be a query and \( Q_{\text{sub}} = \text{op}(Q'_{\text{sub}}) \) is a subquery of \( Q \). icols(op) is a set of attributes \( E \subseteq \text{SCH}(Q) \) such that \( Q \equiv Q_{\text{sub}} \sim \Pi_{\text{icols(op)}}(Q_{\text{sub}}) \).

**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 \( op \). For reasons of space we only present some inference rules for the EC property. The remaining inference procedures and proofs of their correctness are given in our accompanying technical report [21].

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

\[
E(\text{EC}) = \{ E \cup E' \mid E \in \text{EC} \land E' \in \text{EC} \land E \cap E' \neq \emptyset \land E \neq E' \} \\
\cup \{ E \mid E \in \text{EC} \land \text{EE} \in \text{EC} : E \neq E' \land E \cap E' \neq \emptyset \}
\]
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 \( \Pi_A(R) \), we can delay the computation of \( b \) after \( \diamond \) has been evaluated. Rule (2) since \( \text{keys}(R) \neq \emptyset \), 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

\[ a \subseteq \text{SCH}(\Diamond(\Pi_A(R))) \land b \notin \text{ocols}(\Diamond(\Pi_A(R))) \]

\[ \Diamond(\Pi_{A,a \rightarrow b}(R)) \rightarrow \Pi_A(\Diamond(\Pi_{A,a \rightarrow b}(R))) \]

\[ A = \text{ocols}(R) \]

\[ R \rightarrow \Pi_A(R) \]

\[ x \notin \text{ocols}(\omega_{f(a)}(R)) \]

\[ \omega_{f(a)}(R) \rightarrow R \]

\[ \frac{\text{set}(\delta(R))}{\delta(R) \rightarrow R} \]

\[ \frac{\epsilon_1 = \text{if } \theta \text{ then } A + c \text{ else } A}{\Pi_{e_1, \ldots, e_m}(R) \rightarrow \Pi_{A + i f \theta \text{ then } c \text{ else } 0, e_2, \ldots, e_m}(R)} \]
tuples are returned by the operator is independent of input multiplicities. Since \(\text{set}\left(\delta\left(R\right)\right) = \text{true}\), we know that there exists \(\text{op}' = \delta\) with \(\text{op} \sim \text{op}'\) and \(\forall \text{op}'' : \delta\left(R\right) \times \rightarrow \text{op}'' \sim \text{op}'\) such that \(\text{op}'' \notin \left(\gamma, \omega\right)\). All operators except for \(\gamma\) and \(\omega\) are insensitive to duplicates. Thus, the set of tuples in the input of \(\text{op}'\) is not affected by the rewrite \(\delta\left(R\right) \rightarrow R\). While the multiplicities of these tuples can be affected by the rewrite, the final result of \(\text{op}' = \delta\) is not affected. Rule 4. Suppose \(A = i\text{cols}\left(R\right)\), by definition 3 we get \(\overline{R} = \Pi_A\left(R\right)\). Rule 5. From \(x \notin i\text{cols}\left(\omega\left(f\left(a, \ldots, x\right)\right)\right)\) follows \(Q[\omega\left(f\left(a, \ldots, x\right)\right) \leftarrow \Pi_i\text{ch}(R)\left(\omega\left(f\left(a, \ldots, x\right)\right)\right)] = Q\). Based on the definition of \(\omega\) it follows that \(t^n \in \Pi_i\text{ch}(R)\left(\omega\left(f\left(a, \ldots, x\right)\right)\right) \rightarrow t^n \in R\). Thus, \(Q[\omega\left(f\left(a, \ldots, x\right)\right) \leftarrow R] = Q\). Rule 6. Let \(e'_1 = A + \text{if } \theta \text{ then } c \text{ else } 0\). We distinguish two cases: 1) if \(\theta\) holds, then both \(e_1\) and \(e'_1\) evaluate to \(A + c\); 2) otherwise 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 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 [12] for details). For instance, \(\Pi_{b,\text{sum}(a)}(R)\) with \(R = (a, b)\) would be rewritten into \(\Pi_{b,\text{sum}(a),P(a),P(b)}(\gamma \left(\text{sum}(a)\right) R) \Delta \Pi_{b \rightarrow b', a \rightarrow P(a), b \rightarrow P(b)}(R)\). Alternatively, the aggregation can be computed over the input with provenance using the window operator \(\omega\) by turning the group-by into a partition-by. The rewritten expression is \(\Pi_{b,\text{sum}(a),P(a),P(b)}(\gamma \left(\text{sum}(a)\right) R) \Delta \Pi_{b \rightarrow b', a \rightarrow P(a), b \rightarrow P(b)}(R)\). The Window method has the advantage that no additional joins are introduced. However, as we will show in Sec. VI-B, 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] 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 \(\theta_i\) denote the condition (WHERE-clause) of the \(i^{th}\) update. Every tuple updated by the transaction has to fulfill at least one \(\theta_i\). Thus, this set of tuples can be computed by applying a selection on condition \(\theta_1 \lor \ldots \lor \theta_n\) to the input of reenactment. The alternative called HistJoin uses time travel to determine based on the database version at transaction commit which tuples were 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 \(\theta_1 \lor \ldots \lor \theta_n\) 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 \textsc{hasMorePlans}) or until a stopping criterion has been reached (function \textsc{continue}). In each iteration, function \textsc{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 \textsc{makeChoice}. The resulting plan \(P\) is then costed. If the cost of plan \(P\) generated in the current iteration is less than the cost \(T_{\text{best}}\) 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 \textsc{genNextIterChoices}. The difference between our CBO algorithm and standard CBO is the implementation of the \textsc{generatePlan} and \textsc{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. VII-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 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 \textsc{continue}, \textsc{genNextIterChoices}, and \textsc{makeChoice} functions. The default setting guarantees that the whole search space will be explored (\textsc{continue} returns true). Our CBO algorithm keeps track of how much time has been spent on optimization so far (\(T_{\text{opt}}\)) 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 through 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 \textsc{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) 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. 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).

```python
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 taken. In our example, we would use the Window method if 0 is returned.

A code fragment containing a call to MAKECHOICE may be executed several times during one iteration. 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 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 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. 3 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.

Algorithm 2 Default MAKECHOICE Function

```python
1: procedure MAKECHOICE(numChoices)
2: if len(pnext) > 0 then
3:   choice ← popHead(pnext)
4: else
5:   choice ← 0
6: pcurs ← pcurs || choice
7: nops ← nops || numChoices
8: return choice
```

Algorithm 3 Default GENNEXTITERCHOICES Function

```python
1: procedure GENNEXTITERCHOICES( )
2: pnext ← pcurs
3: for $i \in \{\text{len(pnext)}\ldots,1\}$ do
4:   $c \leftarrow $ popTail(pnext)
5:   nops ← popTail(nops)
6: if $c + 1 < \text{nops}$ then
7:   $c \leftarrow c + 1$
8:   $pnext \leftarrow pnext || c$
9: break
10: pcurs ← []
11: nops ← []
```

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 $O(d)$ space. As long as we know the path taken in the previous iteration (represented as a list of choices as shown in Fig. 3) 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 $p_{curs}$ 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 $p_{curs}$ that ends in the new choice to be taken. If following $p_{next}$ leads to a path is longer than $p_{next}$, then after making $\text{len}(p_{next})$ choices the first option should be chosen for the remaining choice points.

Example 4. Reconsider the plan tree shown in Fig. 3 and assume the plan created in the previous iteration is $[0,1]$. The 2nd choice point that was hit does not have any more options, but the 1st one has one additional option. That is, to reach the next leaf node to the right of $p_{curs} = [0,1]$, we should choose the other option $p_{next} = [1]$. This option leads us to a path that is longer than $\text{len}(p_{next}) = 1$, 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., $[0,1]$ denotes a list with elements 0 and 1. We use $[]$ to represent an empty list. Let $L \leftarrow L || e$ denotes appending element $e$ to list $L$. Functions popHead(L) and popTail(L) remove and return the first (respectively last) element of list $L$. 

---

Fig. 5: Plan space tree example
The makeChoice Function. Algorithm 2 shows the default makeChoice function. If possible we pick the next predetermined choice from list $p_{next}$. If list $p_{next}$ is empty, then we pick the first choice (0). In both cases, we append the choice to the current path and the number of available options for the current choice point is appended to list $n_{ops}$.

Determining Choices for the Next Iteration. Algorithm 3 determines which options to pick in the next iteration. We copy the path from the previous iteration (line 2) and then repeatedly 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 2 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 $\tau$ 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 (which we call 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 $\tau$ 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 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 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 with the hope to discover a better plan. The following 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: stop optimization once $T_{best} \leq \frac{T_{opt}}{2}$.

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 23 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. Importantly, our PATs can easily be integrated with an optimizer that applies cost-based transformation.

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 [24] and nested collection models [10]) 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 DBNotes 7, Trio 1, ORCHESTRA 17, Perm 12, LogiBox 14, ExSPAN 26, and GProM 6 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. Optimiza-
tion of provenance has mostly focused on minimizing the
storage size of provenance. Chapman et al. [8] introduce sev-
eral 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 pro-
venance (full provenance is propagated alongside the tuple) was
introduced in the context of declarative networking [20]. This
idea of storing just enough information to be able to reconstuct
provenance through instrumented replay, has also been adopted
for computing the provenance for transactions [6], [5] and in
the Subzero system [25]. Subzero switches between different
provenance storage representations in an adaptive manner to
optimize the cost of provenance queries. Amsterdamer et al. [5]
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 exe-
cution time. All experiments were executed on a machine with
2 AMD Opteron 4238 CPUs, 128GB RAM, and a hardware
RAID with $4 \times 1$TB 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 opti-
mization 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 [5] (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 his-
tory, e.g., $H10$ represents 10\% history (100K tuples). DBLP: 
This dataset consists of 8 million co-author pairs extracted from DBLP [http://dblp.uni-trier.de/xml/].

Simple aggregation queries. This workload computes the
provenance of queries consisting solely of aggregation opera-
tions using the instrumentation technique based on the rewrite
rules pioneered in Perm [12] and extended in GProM [6].
An aggregation query consists 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 select 11 queries from the 22 TPC-H
queries to evaluate optimization of provenance capture
for complex queries. The technique [11] 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 [5]
to compute provenance for transactions. The transactional
workflow is run upfront (not included in the measured execu-
tion time) and provenance is computed retroactively. We vary
the number of updates per transaction, e.g., $U10$ is a trans-
action 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 [22] to trans-
late a relational encoding of provenance (see Sec. IV) 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: 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 $Q_{i,j}$ to denote the $j^{th}$ why question for query $Qi$.

B. Measuring Query Runtime

Overview. Fig. 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
Fig. 12: Datalog Provenance

Fig. 8: 10GB

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

Fig. 7: 1GB SimpleAgg Runtime

Fig. 9: Runtime TPC-H - 1GB

Fig. 11: Provenance Export

Fig. 13: Transaction provenance - runtime and overhead

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

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

Fig. 16: Total workload runtime for transaction provenance

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
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% 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 ~14 compared to CBO. These results also confirm the critical importance of our PATs since deactivating these transformations increases runtime by a factor of ~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 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.7 and 8 show the results for individual queries. Note that the y-axis is log-scale. Activating Heu 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 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 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 Join+NoHeu. 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 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 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 (H100, H110, 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. Heu outperforms NoHeu, because Heu determines that some of these window operators are redundant and can be removed (PAT rule [9]). 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 ~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 measured where time is spent during CBO and have determined that the majority of time is spent 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.

**TPC-H Queries.** In Fig. [14](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. [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-B]. We also combine these strategies with our adaptive (adp) heuristic that limits time spent on optimization based on the expected runtime of the best plan found so far. Fig. [18](right) 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 study provenance-specific algebraic transformations (PATs) and instrumentation choices (ICs), i.e., alternative ways of realizing provenance capture. Our experimental evaluation demonstrates that our optimizations improve performance by several orders 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**


