Compact, Tamper-Resistant Archival of Fine-Grained Provenance

Nan Zheng
University of Pennsylvania
nanzheng@cis.upenn.edu

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

ABSTRACT

Data provenance tools aim to facilitate reproducible data science and auditable data analyses, by tracking the processes and inputs responsible for each result of an analysis. *Fine-grained* provenance further enables sophisticated reasoning about why individual output results appear or fail to appear. However, for reproducibility and auditing, we need a *provenance archival system* that is *tamper-resistant*, and efficiently stores provenance for computations computed over time (i.e., it compresses repeated results). We study this problem, developing solutions for storing fine-grained provenance in relational storage systems while both compressing and protecting it via cryptographic hashes. We experimentally validate our proposed solutions using both scientific and OLAP workloads.

PVLDB Reference Format:
Nan Zheng and Zachary G. Ives. Compact, Tamper-Resistant Archival of Fine-Grained Provenance. PVLDB, 14(4): 485 - 497, 2021.
doi:10.14778/3436905.3436909

1 INTRODUCTION

The need to facilitate verifiable, reproducible analyses has driven the development of many data provenance capture and management tools. Today provenance is used not only to capture, reproduce, and certify scientific results, but increasingly it serves a similar role in audit logging in the enterprise. Provenance management tools must ultimately provide two types of services: (1) capturing data provenance for common computations in an appropriate *provenance data model*, and (2) providing a queriable, auditable, tamper-resistant *provenance-augmented data archive* for analysis results.

Provenance capture has been addressed over the past 15 years in the scientific computing and database communities. File-and-program level ("workflow") provenance capture [7, 25, 28] is sufficient to re-run scientific computations to reproduce files. Conversely, the database community has developed techniques to track provenance at the record level ("fine-grained" provenance) [11, 19, 32], tracing through relational and other operators. The benefits of the latter case are that data scientists can *reason about why specific answers exist* [12] or *explain why two workflow runs produce different answers* [32]. Enterprises can also trace fine-grained provenance to verify compliance with the GDPR data protection laws, by showing whether an analysis result actually uses private records.

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436909

Provenance storage has also been studied, both in terms of the provenance representation (e.g., PROV-DM [8], provenance semirings [19]), as well as encoding in a graph or relational DBMS. Early fine-grained provenance systems developed middleware over relational DBMSs, and rewrote queries and updates to automatically capture provenance [16, 18, 23]. Later work investigated how query engines themselves could be augmented to capture *semiring provenance* [19] as part of the evaluation of relational algebra operators [2, 30, 32], enabling a variety of optimizations. Most recently, the work of Lee and colleagues [23] considered how to rewrite or *factor* relational algebra computations to more efficiently store provenance for individual queries.

However, two key aspects of provenance *archival* have been unaddressed. First, in an archival system, we may need to store many analyses computed from the same input datasets. Since provenance is often bigger than the analysis results, *compression* of repeated provenance is highly desirable. Second, if provenance is to be used to certify results for an audit record, it must be *tamper-resistant*. While recent work on provenance for blockchains [24, 27, 31] and networks [33] develops self-certifying techniques, these approaches do not naturally extend to fine-grained provenance.

In this paper, we focus on how to create a *tamper-resistant archive* of *data analysis results and their fine-grained provenance* generated over time, to enable reproducibility, querying of individual records' provenance, and verification of authenticity. We develop techniques for *derivation-based compression*: different computational analyses often share computational structure, and may be computed over common data sources or over input records. In such scenarios, we exploit structure and repetition across queries and subqueries. (We combine this with value-level compression within the storage system [1].) We consider queries expressed within the relational algebra, augmented with user-defined functions. This encompasses OLAP-style queries [30], as well as common ETL and gene sequence matching [32] operations such as pattern-based extraction, approximate match, and top-k selection.

Example 1. *Consider the following SQL query:* `SELECT * FROM REF NATURAL JOIN INPUT I WHERE name = 'Example'`. *This might be converted into the simple expression:*

$$\sigma_{name='cat'}((REF) \bowtie (INPUT)))$$

Suppose input relations R and S are as follows (initially disregard the "prov" column):

R(***prov***	***name***	***code)***
r_1	*cat*	*f catus*
r_2	*wolf*	*c lupus*
⋮		

S(***prov***	***code)***
s_1	*c lupus*
s_2	*f catus*
⋮	

We get a final tuple (*cat*, *f catus*), *whose provenance is based on the tuples whose prov column includes* r_1 *and* s_2. *More precisely, if* r_1, s_2

are the provenance of the respective input tuples, then the provenance of the output involves joint use *of r_1, s_2 — which we represent as a provenance expression $r_1 \cdot s_2$ [19]. We would like to store this tuple and its accompanying provenance in a form that can be consulted for reproducibility.*

Example 2. *Now suppose we run a second query over the same input data, this time projecting only the code field:*

$$\pi_{code}(\sigma_{name='cat'}((REF) \bowtie (INPUT))))$$

This second query shares a subexpression with the first. Its output and provenance might be derived from those computed for the first query — this is what we mean by derivation-based compression.

Example 3. *Finally, suppose we want to publish the results and their provenance, ensuring ure the source data and the provenance steps have not been tampered with. We could share the data and provenance with cryptographic hashes establishing authenticity.*

We develop and study techniques for efficiently capturing fine-grained provenance while archiving results, in a way that (1) takes advantage of repeated computation for efficiency, and (2) uses cryptographic hashes that allow the provenance to be self-certifying. Our encoding schemes enable low-overhead storage as well as fast provenance queries, and we consider trade-offs between speed and cryptographic security. Our provenance archival service is suitable for maintaining a verifiable and reproducible scientific record — perhaps the most important use case of provenance in data science — and we also support authenticated audit logs for any generated reports. We support tracing individual results not only for reproducibility but also for reasoning about the effects of change. Our methods emphasize provenance integrity-checking given *digests* of the input data, rather than requiring full access to the data itself.

Threat model. We assume a trusted execution environment, in which our own code is not tampered with, and seek to defend against:

- Users who generate an analysis, and seek to tamper with the result — changing provenance records for specific (1) input, output, or intermediate results, (2) query operations run, (3) versions of the source and/or binary code executed.
- Attackers who, given a published analysis result, wish to substitute a new result, or provenance for such a result.

The focus of our paper is on maintaining the integrity of individual results – but our techniques naturally generalize to maintaining a tamper-resistant directory of all analysis results produced, thus also preventing an attacker from deleting existing results. We rely on the use of digital signatures to establish the creator of a result, which would prevent attackers from forging new results.

Our contributions are as follows:

- Provenance encoding techniques that ensure results computed *using the same algebraic expressions over the same input data* will be stored exactly once, irrespective of the number of queries; and which are *self-certifying* (tamper-resistant).
- Novel algorithms to compare provenance expressions for equivalence, and explain where computations diverge.
- A cost model for provenance storage, as well as an understanding of when the optimal query evaluation plan also results in the optimal storage scheme.
- An implemented tamper-resistant provenance archival layer that extends the open-source PROVision [32] system.
- Performance analysis over a variety of workloads taken from ETL and scientific data management.

Section 2 summarizes our relationship to related work. Section 3 provides background to our approach and Section 4 describes how we encode fine-grained provenance. Section 5 describes our PROVision implementation, and Section 6 provides an experimental evaluation of our approach. We conclude in Section 7.

2 PRIOR WORK

Fine-grained data provenance has been widely studied [11]. We adopt the provenance semiring model [18, 19]: each tuple is annotated with expressions comprised of *tokens* representing the input tuples, the product operator ($\cdot$) representing joint use of inputs to produce a result, and the sum operator (+) representing alternate derivations of the same value. Extensions consider grouping and user-defined functions [3, 32].

Early provenance systems [16, 21] integrated provenance into existing DBMSs, generally by *rewriting* queries to store provenance as auxiliary information. Lipstick [2], Smoke [30], and PROVision [32] studied how to incorporate provenance tracking directly into custom query engines. The order of evaluation of results affects provenance expression size. Lee and co-authors [23] reduce provenance storage by creating more efficient *factorized* query expressions [29].[1] Their PUG system *factors* provenance into a *d-tree* representation [29] before storing it. Similarly, Bao et al. [6] develop strategies for factoring out provenance storage for common query expressions. In contrast to such work, we focus on minimizing query overhead (the common case) and show that our approach is optimal under certain (simplifying) assumptions about cost; we develop mechanisms for "compressing" and authenticating the storage; we support more expressive operations including aggregation.

The encoding of fine-grained provenance has also been studied extensively. Perm [16] uses *witness lists* to capture, for each output, the set of input tuples used; it duplicates the output tuple if there are multiple witnesses. It can annotate results with their associated SQL statements. Orchestra [21] maintains a derivation graph using semiring provenance tokens. Chen et al. [10] targets network datalog execution, and develops a hash-based encoding scheme for derivation trees, which eliminates intermediate nodes. Anand et al. [4] encode provenance for updates to hierarchical data.

We apply cryptographic hashing to provenance. This has been considered in network diagnosis [33], where it was used to sign distributed system events in a log. Additionally, several efforts have integrated provenance into smart transactions in a blockchain [27, 31] and into storing coarse-grained provenance [24]. To the best of our knowledge, we are the first to consider cryptographic hashing as both a security and sharing mechanism for fine-grained provenance using extensions of the semiring model.

3 BACKGROUND AND APPROACH

Fine-grained provenance capture involves recording, for each tuple derivation, the relationship between the inputs (which are each assigned unique *provenance tokens* as IDs) and the output (for which

[1] Factorization has also been applied to workflow provenance [9].

we can also assign tokens). We begin this section with an overview of the formal model. For provenance archival, our goals are twofold: (1) create provenance tokens in a way that ensures the same expressions result in the same tokens, i.e., repeated computations receive identical token values; (2) create provenance tokens in a way that prevents forgery or tampering. This section reviews the work providing foundations for our approach, and in Section 4 we discuss our novel algorithmic implementation. We assume the query processor computes provenance as it derives results [30, 32].

3.1 Provenance Semiring Expressions

We adopt the provenance semiring model [18, 19], which *annotates* tuples with *provenance polynomial expressions* satisfying the properties of an algebraic semiring. Relational algebra expressions that are equivalent under bag semantics have provenance polynomials that are equivalent. Here, base tuples are annotated with unique IDs called *provenance tokens*. As relational algebra operators are applied, we annotate each derived tuple with an expression over the provenance of the input tuples. Operators in the expression may be the product $(\cdot)$ representing joint use of inputs to produce a result, or the sum operator $(+)$ representing alternate derivations of the same value. If we let $P[t]$ designate the provenance expression for tuple t, and look at inputs and outputs as paired tuples and their provenance — then the rules are:

Expr.	Inputs	Outputs
$\sigma_\theta(T)$	$(t, P[t])$	$(t, P[t])$
$\Pi_a(T)$	$(t, P[t])$	$(t, P[t])$
$T_1 \cup T_2$	$(t_1, P[t_1])$, $(t_2, P[t_2])$, $t_1 \in T_1 = t_2 \in T_2$	$(t_1, P[t_1] + P[t_2])$
$T_1 \bowtie T_2$	$(t_1, P[t_1])$, $(t_2, P[t_2])$, $t_1 \in T_1$, $t_2 \in T_2$	$((t_1 t_2), P[t_1] \cdot P[t_2])$

Provenance management systems [16, 18, 21] often encode the polynomials in relational form (in effect as expression trees or graphs). Under this model, while multiple provenance expressions are equivalent, the order of evaluation of the relational algebra expressions notably affects their overall size.

Extensions to the semiring model support aggregation functions [3], as well as *user-defined* aggregate and extraction functions [32]. Since this paper is focused more on the *structure* of provenance expressions rather than semantics, we briefly sketch how this works here, and refer the reader to [32] for details.

For each UDF, we introduce a *function symbol* in the provenance polynomials. A given UDF, e.g., information extractor, typically extracts data from within some datatype-specific *location specifier* within a field (e.g., a substring within a string attribute). Finally, a series of function- and datatype-specific *equivalence rules* may define how different provenance expressions are equivalent.

Example 4. *Consider a gene sequence matching workflow [13], specified using the relational algebra and UDFs. This workflow can be captured using physical query operators, as follows: (1) file-scan operators over input text files, reading from them source and reference genome relations S and R; (2) a UDF pfx that extracts a prefix from the sequences in S and R; (3) a join matching the prefixes extracted from the sources; (4) a filter expression θ which returns results exceeding some overall similarity criterion. We might encode this as:*

$$\sigma_\theta(udf_{pfx(refseq)}(scan(R)) \bowtie udf_{pfx(seq)}(scan(S))))$$

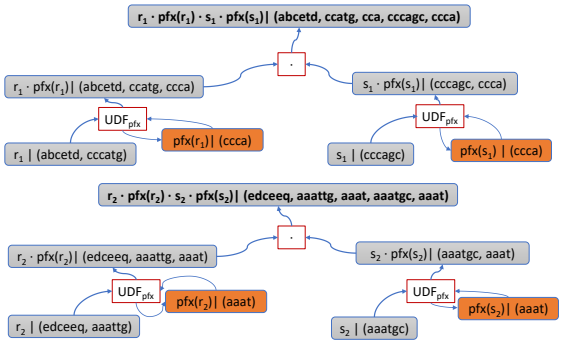


Figure 1: Provenance graph for Example 4 query.

where the UDF calls function pfx over each tuple. Suppose input relations R and S are as follows:

R(***prov***	***name***	***refseq)***
r_1	*abcetd*	*cccatg*
r_2	*edceeq*	*aaattg*
⋮		

S(***prov***	***seq)***
s_1	*cccagc*
s_2	*aaatgc*
⋮	

Suppose function pref returns the first 4 characters of the sequence; then for tuples r_1 and s_1 we will get substring ccca and for tuples r_2 and s_2 we will get prefix aaae. The two pairs of tuples will join. Assuming the threshold is met, we get final tuples ('abcetd','cccatg','cccagc') and ('edceeq','aaaedf','aaaeij').

Here, join operators correspond to product operations $(\cdot)$ in the provenance polynomials. Following [32], user-defined functions are a kind of joint use *(combination or Cartesian product) of the input source tuple values from s combined with the results of the UDF, $f(s)$: $s \overrightarrow{\bowtie} f(s)$[2]. Thus, each result tuple is annotated with a new provenance expression containing function f', the input provenance, and additional information (a* location specifier *[32]) identifying the data used by f to return its result.*

For example, if we take tuple $R(edceeq, aattg)$ with provenance token r_2 and apply the UDF pfx over it, the result will be a tuple $(edceeq, aattg, aaat)$ with provenance expression $r_2 \cdot pfx(r_2)$ where pfx' represents a predicate extracting a new token ("location specifier") based on r_2 and the semantics of function pfx.

3.2 Computing Query Results with Provenance

Section 5 describes our implementation, but at a high level, PROVision's relational query operators not only compute a stream of output tuples from the input tuples; but annotate such tuples with provenance expressions derived from the input tuples' provenance [32]. As query results are computed in memory, provenance expressions are encoded as objects passed by reference, so internally each tuple's associated provenance annotation is an object representing a rooted expression tree (consisting of semiring operations, function symbols, and provenance tokens). Any intermediate node links to the provenance annotation nodes on the input tuples, and so on, back to the provenance over the source data.

[2]The *dependent join* [15] $\overrightarrow{\bowtie}$ models parameter passing to a function as a join.

Example 5. *See Figure 1 for the provenance graph corresponding to Example 4. Each base tuple (r_1, r_2, s_1, s_2) is fed to the UDF pfx, producing a result (orange) that is combined with the input tuple. The results are joined $(\cdot)$ to produce the boldfaced outputs.*

This paper considers how to persist the in-memory structure in relational storage, (1) allowing sharing of graph nodes *across queries and query subresults*, (2) enabling rapid lookups of subexpressions, and (3) cryptographically certifying the provenance. To do this, we develop a strategy for assigning node IDs, such that two provenance subgraphs have the same node if they are isomorphic, and the node IDs include a cryptographic hash of the subgraph contents.

4 ENCODING PROVENANCE GRAPHS

We now consider how to take provenance that is computed during query processing, and map it to persistent storage. As described in the previous section, the provenance for a running query is encoded in-memory as a *directed acyclic graph*, where nodes represent expressions and implicitly have edges to their subexpressions, captured as object references. The basic query processing model of Section 3.2 creates the graph on a per-query basis.

As a provenance graph node is being mapped to disk, intuitively we assign its identity as a hash of its provenance polynomial expression. This allows us to take a provenance polynomial expression, and quickly look it up (create a new node if it does not exist). Edges are simply binary relations between hashes.

We outline our approach, then describe our storage scheme and its tamper resistance. We conclude the section by looking at the conditions under which we expect the provenance graph to be space-efficient if the query optimizer is minimizing execution cost.

4.1 Provenance Encoding for Archival

Our goal is to support the storage of many derived query results, each accompanied by semiring provenance expressions. We assert that such an archival system should support *subexpression sharing* for efficiency, and, as a means of enabling auditing, it must also enable *certification of authenticity* that results and their inputs have not been forged or tampered with.

Cross-query sharing. If we define a test for semantic equivalence of source (Section 4.1.1), then if two queries perform *the same algebraic query expression* over the same source tuples, we can consider the derivation tree (the results and the provenance) to be the same. Under those assumptions, we should store the subtree for such an expression (a derivation) once, and include it in the provenance of both queries (derivation-based compression). We can maintain a single provenance directed acyclic graph that shares common subexpressions — resulting in more efficient storage. This requires a *compositional* encoding scheme for provenance expressions, in which a provenance expression can be composed from other provenance subexpressions through factoring and substitution.

Tamper resistance and threat model. We seek for provenance structure that is self-certifying, protecting against attacks on the integrity of the data or code used in a result. We assume a trusted query execution environment, and that standard cryptographic signatures can be used to prevent forged results. We assume a model in which the source data and binary operations (UDFs) will remain available in archival form on the Internet, accompanied by signatures — e.g., through GitHub, public repositories, or archived storage — but that a derived result and its associated provenance structure (a) might be tampered with in order to manipulate the record, or (b) might be alleged to have been derived from a *different* source or using *different* code. We do not assume that standard cryptographic hash functions are collision-free, but assume that we will use a combination of cryptographic hashes that makes it highly improbable that an aspect of the provenance can be modified without affecting a hash value in a tamper-evident fashion. We address both problems through the careful choice of *provenance tokens* and cryptographic hashing.

4.1.1 Encoding Source Tuple Provenance. A first question is how to ensure the provenance structure "certifies" the value of the inputs. To do this, we assign values to provenance tokens with cryptographic signatures of the data. In fact, there are a number of subtleties in how we might formalize this, all having to do with the *context* of a data record: should provenance consider **where a record appears** (considering, e.g., the file's full contents, the file's name or semantic description, the record's position) or just **what its value is**? We consider three formulations which preserve increasing amounts of context:

(1) *Value-equivalent*: only the data matters, so we create a token by hashing the record.
(2) *Origin-equivalent*: here we consider data to be different if it appears in a different place (within the file, if there are duplicate records; or across files). We create a token by hashing the file URI and timestamp, plus the location of the data.
(3) *Content-and-origin-equivalent*: the provenance of a tuple further depends on the specific file *and version* in which it appears. Here, we create a token by hashing a digest of the file contents, file URI, and the location of the data.

Other definitions are possible, but we feel these three definitions represent the majority of use cases encountered in data science.

Note that the choice of contextual information not only provides different levels of detail about the source, but affects when two provenance derivations are considered equivalent. As we include more contextual information in the token, there are fewer cases when source results are considered the same.

4.1.2 Encoding Derived Expressions. Next we consider provenance expressions for derived results. Generally, one would need to reason about *algebraic equivalence* to determine if two provenance expressions are equivalent. However, a sound-but-incomplete scheme for testing for algebraic equivalence is to look at whether the provenance expressions are structurally equivalent (i.e., the same relational algebra expression over the same inputs) with the same inputs. We develop a scheme that derives the provenance token in a deterministic way that ensures the *same subexpression* evaluated over the *same input data* always has the same value. This property allows us to (1) store the provenance token and its derivation exactly once in memory, and (2) quickly determine that two expressions are identical if their tokens are identical. We can achieve this by applying strongly collision resistant (cryptographic) hash functions to the provenance expressions.

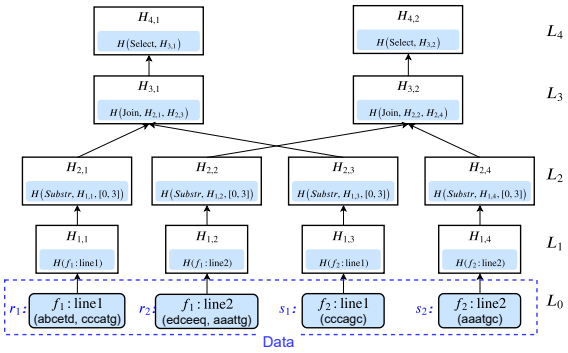


Figure 2: Merkle tree for Example 4 query.

A natural question is when two operations are equivalent. For relational algebra operations, the physical operators implementing the same logical operator (e.g., different join implementations) are considered equivalent. For user-defined functions, this gets more complex: our implementation assumes that there exists a digest for each function, and that functions are identical if their digests are identical. Of course, if the functions have dependencies to external libraries, or if different runtime conditions affect operation, one would need to generalize this approach. Such approaches require substantial engineering but would fit naturally into our approach (the digest would include all relevant dependencies).

Using cryptographic hash functions as the basis of provenance expression identification provides us with a second opportunity. We can produce an encoding that is self-certifying, in the sense that we can verify that an output was produced through a particular derivation from a given input, and any tampering with the structure or data would not satisfy the hash. To do this, we adopt the Merkle tree [26, 31], a data structure that has recently been popularized in blockchain protocols [31] and used for auditing and accountability [20]. (Our Merkle tree implementation supports variable arity for intermediate nodes.) A Merkle tree in effect uses recursive hashing: an intermediate node is assigned the hash of its children's hashes, and so on to the leaf level.

For a provenance expression, we derive a provenance token as the *cryptographic hash* of the root operator and any inputs (which, for intermediate nodes, are themselves recursively computed as cryptographic hashes of their inputs, all the way back to the leaf nodes, whose tokens are computed as cryptographic hashes of the original input tuples as in Section 4.1.1). This both ensures the provenance is tamper-resistant, and also guarantees that a provenance token derived from the same expression and inputs will always have the same value[3].

Example 6. *Figure 2 shows the Merkle tree of Example 4. The leaves are the records from the reference genome file f_1 and source genome file f_2. Each line of file f_1 encodes a tuple from relation R, and similarly for f_2 and relation S. Each such input will be assigned a provenance token, here computed as the hash of the filename and position (origin-equivalent). The first operator,* extract prefix, *will take the input tuples and generate a token (e.g., $H_{2,1}$ as an example) for each output, by hashing the operator name ($extract_{pref}$) and input token ($H_{1,1}$). Each operator assigns a provenance token to its output, generated by hashing the unique value of the operator type, parameters and input tokens. The token for each output takes the hash of its child hash values (augmented with additional information), forming a Merkle tree structure of the hash tokens.*

In PROVision, we consider two ways of protecting provenance with Merkle trees. The simplest approach is to use large enough cryptographic hashes to make it effectively impossible to find a hash collision and forge a result. Here provenance can be verified without access to the input data. A second approach is to use a smaller hash function in the Merkle tree, where collisions are incredibly improbable but not computationally intractable to find — but then to restrict the range of possible values an attacker can use, by using input data which is permanently available and signed.

4.2 Storing the Provenance Graph

Our goal is to encode the graph on disk in relations, in part because this is broadly applicable across a wide range of DBMSes and applications. Our on-disk representation is based on a generalization of a labeled, ordered *edge relation* for a directed graph. We focus on compression at the *logical-level*, relying on existing DBMS physical-level data compression to complement our work. Here, we have an *n-ary* relationship between input expressions (by token), an operator, and a result. Conceptually, the schema of the *PTable* ("provenance table") relation is:

$$\langle input, label, index, \textbf{derived} \rangle$$

where *input* is the ID of an input node, *label* defines the algebraic operation or versioned UDF, *index* represents the index of the ordered edge with respect to the derived node, and *derived* represents the ID of the newly derived node.

The *derived* value is generated by hashing the derived tuple's key fields, the operator that produced the tuple, and the list of input tuples' tokens. It is deterministically computed based on the tuple value and provenance polynomial expression. However, even strong cryptographic hash functions may produce collisions. Thus, the actual, unique node identifier we use is the *RID (unique row ID in the storage system) of the first edge* stored for this derived node. Each of the *input* IDs in the record above is in fact the RID of the record defining the input tuple's derivation, as opposed to the actual hash value for that tuple's derivation.

4.2.1 Basic Relational Algebra. As described in Section 4.1.1, the provenance nodes for raw input tuples are given node IDs derived from the tuple values and the appropriate level of *context*. The tuple keys and context are stored as the node identifier, and the remaining fields can be stored as node content. Selection and projection do not create new nodes in the provenance graph. Join $(\cdot)$ and union $(+)$, as binary operators, yield two provenance derivation edges.

Example 7. *The join output of Example 1 is represented as two records, one for each input to the join $(\cdot)$ operation:*

$$PTable\langle r_{1_{RID}}, (\cdot), 0, h(r_1, r_2, 'Join') \rangle$$
$$PTable\langle s_{2_{RID}}, (\cdot), 1, h(r_1, r_2, 'Join') \rangle$$

[3]Note by "same expression" here, we refer to an algebraic expression with the same ordering and structure.

where $h(r_1, r_2, 'Join')$ represents the cryptographic hash of the expression, $r_{1_{RID}}$ represents the unique row ID of the 0th derivation record for r_1, and $s_{2_{RID}}$ represents the unique row ID of the 0th derivation record for s_2.

Union has an analogous structure.

4.2.2 Aggregate & User-Defined Functions. Grouping and aggregation (whether through standard or user-defined functions) are encoded analogously to join, but an aggregation may be computed over an arbitrary number of rows. Here, given n inputs to some aggregate function f, we will have n rows of the form:

$$PTable\langle r_i, 'f', i, h(r_1, \ldots, r_i, \ldots, r_n, 'f')\rangle$$

where $1 \leq i \leq n$, $'f'$ represents the name of the function, and as before $h()$ represents the function responsible for generating a deterministic node hash.

Table-valued functions. PROVision supports *table-valued* functions, which return multiple result rows that must be joined with the grouping keys. To incorporate this, we add an additional field to the schema, to capture the output row index.

Equivalent-but-not-identical query expressions. Our strategy compresses results produced via the same provenance polynomial expression. This means they must be produced by query subexpressions that (other than selection and projection conditions), follow the same evaluation order. A question we studied was whether we could *rewrite* the subexpressions to equivalent expressions in order to increase sharing (if an existing result used a different evaluation order). Unfortunately, because query expressions are computed bottom-up, sub-expressions are produced before super-expressions, making techniques for rewriting the provenance polynomial from one expression to another ineffective in reducing storage overhead. We thus rely on the fact that the optimal evaluation plan for the same subexpression will be the same, even under minor variations to the selection and projection conditions. In Section 4.4 we discuss how the query optimizer's choice of plans affects the size of the provenance graph.

4.2.3 Lookup and Tracing. The *PTable* table is indexed (1) by row ID, allowing direct random-access lookups to the first derivation of a node, as well as sequential lookups of the remaining derivations; (2) by node hash, allowing lookups by hashing the provenance expressions. The latter is used to determine if there are multiple colliding nodes with the same hash value, or to determine if a subexpression has been previously computed. In the latter case we can *share the subexpression.*

Provenance querying frequently involves *tracing* from a subset of the *PTable* relation P, namely those output nodes that satisfy predicate ϕ, via transitive closure until we get to the input data. Algorithm 1 traverses the edges in P to return those that are part of the subgraph. A second common type of provenance query involves looking for overlap or differences in results (either within or across queries). Here, we can immediately detect commonality by checking the provenance token values as we trace.

Algorithm 1 RetrieveProvenance(P, ϕ)

```
Ret = {e|e ∈ P ∧ φ(e)} {Edges satisfying φ}
Parents = {edge.input|edge ∈ P}
if Parents ≠ ∅ then
    Ret = Ret ⋃ ProvSubgraph(P, λe : e.RID ∈ Parents) { New φ: edge e's RID appears in Parents }
end if
return Ret
```

4.3 Tamper Resistance

Our discussion in this section has largely focused on the use of cryptographic hashing of provenance expressions as a way of generating a (nearly) unique signature for each expression. However, recall from Section 4.1.2 that we actually propose not to precisely hash each provenance expression, but rather to build a Merkle tree to assign a cryptographic hash to it — making it tamper-resistant, and thus useful for maintaining an audit trail, as is needed for a provenance archive.

As in our example of Figure 2, provenance is generated by the PROVision query engine bottom-up, starting with the leaf-level table scan operators. As records are read (and filtered with any pushed-down predicates), their values are cryptographically hashed and added as nodes to our provenance graph edge relation (if an identical node does not already exist). Each node n has a (nearly impossible to forge, but not fully collision-free) cryptographic hash ($n.derived$) and upon insertion receives a unique ID ($n.RID$).

At the next relational operator, we take the derived tuple n' and create a set of derivation edges, each linking to the input tuple(s) (via the unique RID). The cryptographic hash $n'.derived$ is formulated as the *hash of the list of inputs' hash values and any additional parameters, e.g., the derivation operation.* We do not rely on the integrity of the rows in the provenance graph storage system, which could in fact be vulnerable to tampering. Instead, we use the Merkle tree to ensure integrity.

Each time PROVision makes a derivation, it looks in the stored provenance graph to see if the derivation has previously been performed — if so, we simply reuse the node, sharing the representation of the subexpression in the graph. This repeats recursively all the way until the root of the tree (the provenance expression of the tuple output by the query) is computed and stored.

We may want to ensure the integrity not only of individual output records, but of an entire output table, i.e., to certify that no records have been added, deleted, or replaced. Here, we can either attach a digest of the result (as is sometimes done when sharing files), or, more commonly we add one more level to the Merkle tree (hashing the hashes of all records) and use that as a digest.

4.4 Query Optimization and Provenance Size

Prior work [23] has studied *factorized representations* [29] to reduce overall provenance size. Our scenario is somewhat more complex: PROVision attempts to compute results and provenance efficiently, and thus includes a cost-based query optimizer. We *opportunistically* exploit shared subexpressions as they occur.

The size of the provenance graph is determined by the expression used to compute the query. One might develop a dynamic

programming-style cost optimizer to minimize the size of the provenance graph, since space overhead is as easy to estimate as running time. This has some drawbacks, e.g., it would have to rely on heuristics to determine which physical operators to use for each logical step (each results in the same physical storage cost). However, we observe that the provenance graph size is proportional to the amount of intermediate state generated during the query – which is also correlated with execution cost. In this section we seek to understand this relationship, by considering *under which assumptions would a minimum-work query plan, as produced by an optimizer, also produce a minimum-size provenance graph for a given query.*

Intuitively, if we ignore access path selection and the choice of algorithms, the query optimizer cost model emphasizes *minimizing the number of intermediate results*, which minimizes the amount of I/O and work done. Under certain assumptions, the number of "expensive" computations in the query mirrors the number of provenance nodes and edges in the provenance graph.

- Assume all selection and projection operations are maximally pushed down. Suppose we have an index such that *only* those tuples satisfying the selection conditions are retrieved, and we must retrieve an entire tuple to project the desired fields. For a relation R with n tuples, this results in the loading and parsing of n tuples, at some cost c_{load}. If in the provenance graph we only store those nodes satisfying the pushed-down selection conditions, this correspondingly results in the creation of n nodes at storage cost c_{source}.
- Assume a single join implementation, a hash join (or a merge join if data is sorted) with adequate RAM. Given tables R, S with sizes m and n, respectively, a hash join performs $O(m+n)$ hashes during the build and probe stages. It performs $|R \bowtie T|$ Cartesian product operations (c_{join}, which typically are the dominant factor in the cost) to produce the join output tuples. The provenance graph will include $|R \bowtie T|$ new derivations from source tuples with storage cost $c_{product}$.
- Assume only one group-by implementation that calls an aggregate function expressed as a UDF. Then for g groups averaging size n_g, the group-by will call the UDF g times, at cost c_{udf}, each over an average of n_g inputs, resulting in g outputs. The provenance graph will include g nodes representing the groups, linking to n_g input nodes, and producing g result nodes; assume these links take storage cost c_{agg}.

If the above assumptions are met, and we can find a single weight w relating query processing time vs provenance storage time, such that $c_{load} = w \cdot c_{source}$, $c_{join} = w \cdot c_{product}$, and $c_{udf} = w \cdot c_{agg}$, then minimizing the query cost also minimizes the size of the provenance graph. Of course, some of these assumptions may not hold in practice, but nonetheless it is clear that, with a single implementation for each operator, optimizing for query running time is a reasonable heuristic for optimizing provenance graph size.

5 PROVISION IMPLEMENTATION

We now describe our implementation within our custom provenance management and in-memory query/workflow engine, PROVision [32]. PROVision is an execution engine for processing structured data stored in files, serving a similar role to Python with Pandas, Apache Spark, etc. Users will typically write scripts that execute multiple queries in steps, and/or they may compose queries over views. We allow queries to be written both in SQL and in a lower-level, manually-defined, physical algebra form. Both SQL and algebraic queries can call user-defined functions in Java and Python, which is essential for supporting many kinds of operations in scientific data processing, including gene sequence matching as well as ETL-style data cleaning and record linking.

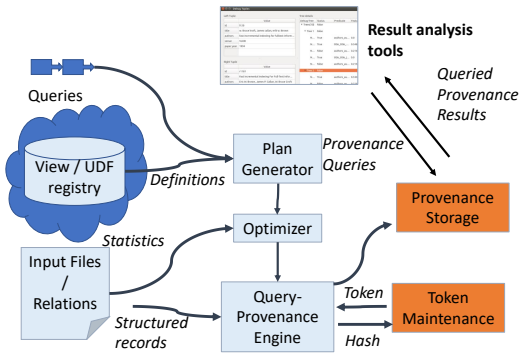


Figure 3: PROVision system architecture.

Our query engine [32] supports standard relational algebra (SPJGU) operations, as well as UDFs, and automatically annotates each tuple with provenance. PROVision allows "eager" computation of a query's output results with provenance, as well as "lazy" computation in which the user may select specific outputs for which he or she wants provenance. This paper focuses on eagerly computing *all* provenance and storing it for archival.

5.1 System Architecture

Figure 3 depicts the modules of PROVision. Given a query containing user-defined functions, PROVision looks up any references to view definitions or user-defined functions in a centralized **Registry**. The **Plan Generator** takes any combination of SQL queries, relational algebra expressions, and (Java and Python) UDFs, and generates a single query plan.

Query optimization. PROVision uses a Volcano-style rewrite-based query **Optimizer** [17] to find a more efficient rewriting of the plan to execute. It consults a data catalog which includes statistics about sources (cardinalities, numbers of unique values) that are cached across query runs. The optimizer's cost estimator uses information about key-foreign key dependencies, as well as input cardinalities, to estimate intermediate result sizes and costs. User-defined functions (which may return table-valued results, in addition to single values or tuples) are included in the search space.

Query execution. The **Query-Provenance Engine** implements a pipelined relational query processor in Java. User-defined functions may be written in Java or Python (via Jython); UDFs use an API to provide detailed provenance information to PROVision [32]. PROVision maintains its own internal representation of provenance as expressions annotating each tuple: As described in Section 3.2, each provenance expression is rooted as an in-memory object, which links by reference to the provenance expressions associated with input tuples, forming a recursive data structure across tuples.

Provenance storage. The focus of this paper is on the rightmost two modules. We leverage BerkeleyDB to persist data within PROVision. To enable tuples and provenance to be stored externally, PROVision uses a **Token Maintenance** system to generate globally unique IDs for provenance nodes as they are derived (described next), and it stores corresponding provenance graph edge information in a **Provenance Storage** system (Section 4.2).

5.1.1 Token Maintenance. At each step of provenance computation, the Token Maintenance module returns a canonical node ID (provenance token) based on the operator or versioned user-defined function being executed, the inputs, and any other parameters (see Section 4.2). Generally, we will directly use a cryptographic hash of the corresponding provenance derivation step, i.e., the root operation list of hashes of the input tokens, and the parameters, as per the Merkle tree structure of Section 4.3). Suppose this is D, in which case the cryptographic hash is $h(D)$. Unfortunately, at scale and for corner cases, we may have to deal with a situation in which two values of D have the same $h(D)$.

Detecting Collisions. We maintain a *hash-input map* (as a B-Tree with a large in-memory cache) from $h(D)$ to the set of possible Ds that hash to the same value. Every time we receive (or wish to look up) a new derivation D, we compute $h(D)$, then look up any matching entries in the hash-input map. If there is no entry, or if $h(D)$ maps to D, then we are collision-free, so $h(D)$ can be used to look up all tuples related to derivation D in the *PTable* relation encoding the graph. We use the Provenance Storage module (described below) to look up the first tuple matching $h(D)$ and return its node ID as a "pointer" to the node entry in the table.

Resolving Collisions. In the event of a collision, $h(D)$ is not a unique key, thus cannot be used to look up or update the hash node. If this is the first occurrence of derivation D, we call the Provenance Storage module to add all tuples related to this derivation to the *PTable*, and retrieve the row ID of the first derivation. In a *collision map* structure, we record a mapping from D to this row ID. We can look up D in the collision map to retrieve and return the row ID, as the identity of the node. We expect collisions to be rare, so the overhead of token maintenance should be as low as possible. The collision map (while persistent) should easily fit into memory. We study the overhead in Section 6.

5.1.2 Provenance Storage. The *PTable* structure described in Section 4.2 is mapped to a B+ Tree. Each derivation, which may involve several edges, is written transactionally. The storage manager retrieves the row ID of the first derivation edge as a unique location for the node. Additionally, the *PTable* relation is indexed by the derivation hash, so it is possible to look for provenance derivations based on the cryptographic hashes at the roots of their Merkle trees. Additionally, we store the persistent state of the Token Maintenance module in B+ Trees. For each of these structures, we rely on buffering and batching to queue up writes, and we rely on buffer pool management to cache frequently accessed values.

5.2 Tamper-Resistance vs. Speed/Space

While cryptographic hashes are nearly collision-free, they impose notable overhead — not just computationally, but also in terms of *space*. Many cryptographic hash functions output 256 or more bits. For PROVision, we looked at the space of cryptographic hash functions and identified the Blake2 algorithm [5] as having a good trade-off of computational speed (it is faster than MD5 and the SHA family) and security (its security analysis indicates it is as secure as SHA3 with a 512-bit digest). Moreover, Blake2, while slower than Java's built-in hashCode(), in practice added negligible overhead to our overall running times. Blake2 is "tunable" to hash digest sizes from 1 byte to 64 bytes, with stronger security guarantees and fewer collisions as we increase the size of the digest. Security trade-offs for the different sizes can be considered based on the probability of a collision [5]; in the next section we empirically look at the *performance* trade-offs (I/O cost, cost of resolving collisions) of different hash sizes.

6 EXPERIMENTAL EVALUATION

Our evaluation of PROVision studies its overhead versus its ability to compress provenance and to certify the integrity of results.

Tasks and queries. Fine-grained provenance has a range of applications, including loading, wrangling, querying, and analysis. We consider simple OLAP queries based on TPC-H, scientific *Genome* sequence matching, and common ETL tasks such as schema matching (*Magellan* [22]) and data cleaning (*DuDe* [14]).

TPC-H: We use the TPC-H data generator with Scale Factor 0.1 (107MB) to generate 8 input tables with a total of 867K rows. We pick single-block SQL queries (Q1, Q3, Q5), avoiding negation. PROVision executes these queries directly over the CSV files, since its engine focuses on outside-the-database tasks. Q1 does a single aggregation; Q3 joins three tables; Q5 joins six tables.

Gene sequence alignment (Genome). Scientists often seek to quantify the genes and related proteins from DNA-sequenced tissue. A query cleans the sequence records (`trim`), `aligns` trimmed sequences against a reference "library" of genes, and finally looks up the genes to determine which proteins are coded. We use components from the STAR alignment toolkit as Python UDFs in PROVision. Our experiments use 145.5M sequences.

Entity matching (Magellan). The Magellan [22] entity matching toolkit provides algorithms for linking records. Magellan includes stages for *blocking* (comparing subsets of record pairs to find an alignment) and *matching* (determining which pairs match above a threshold). Building on example queries provided with Magellan, we link entities between the *ACM* Digital Library (1813 records) and *DBLP* (1780 records).

Data cleaning (DuDe). Another common ETL task involves cleaning records within a data set. The DuDe [14] data cleaning framework searches for pairs of tuples that represent the same real-world object (deduplication). Our experiments use a standard DuDe example, cleaning a compact disc dataset with 9763 records comprised of 107 (possibly null) attributes.

Workload generation and data. We generate workloads as sequences of queries over subsets of data. For the TPC-H queries, we use the standard query generator to create and parameterize queries Q1, Q3, and Q5; we use tables from the data generator.

For our scientific and ETL tasks, we simulate the execution of similar data analysis tasks across time: subsequent queries are similar but may use different UDFs or selection predicates, and some may be over the same input tables whereas others will be over new

data. We define two configuration parameters: the proportion of repeated sub-segments of the input data within a query or across queries (see below for details on data generation) and the similarity between queries in a workload. Each query is generated by parameterizing a template with randomly chosen values, as follows:

- *Genome.* Gene sequence alignment first performs up to 8 trimming conditions. A given query may take any subset of these trimming conditions. For the second step, we choose one of three versions (2.3.0, 2.3.1 and 2.4.0) of STAR.
- *Magellan.* Magellan record linking templates are divided into blocking and matching steps. The template for blocking tests for attribute equivalence, or instead uses rules. The matching stage chooses one of several string similarity measures.
- *DuDe.* DuDe performs duplication detection. As with Magellan, the template randomly chooses a similarity function.

In each workload step, we may apply the query over a different set of input relations sampled from the original dataset. We control both the sizes of the inputs and the amount of duplication (which is important for measuring the benefits of subexpression sharing). Given a source relation with n records, we sample k records uniformly at random. Then starting from each chosen record, we include the next m consecutive rows. We sample with replacement until we reach our target of n' records. This gives us a set of records with a duplicate-record ratio of $(k \times m)/n$.

Environment. Experiments were conducted on an Intel Xeon E5-2630 running at 2.20GHz with 24 cores and 64GB of RAM. Our implementation used the OpenJDK 1.11.0, Python 3.6, PostgreSQL 10.12, BerkeleyDB JE 7.5.11, and the BLAKE2b cryptographic hash implementation. Results are averaged over 5 runs and we present 95% confidence intervals. We experimentally answer the following:

(1) What are the space and time overhead of cryptographic hashing, and of token management? Can we trade off probabilistic security guarantees for space? (Section 6.1.)
(2) How effectively does our scheme compress provenance, both within and across queries? (Section 6.2.)
(3) How effectively does our encoding scheme support tracing and comparing provenance? (Section 6.3.)

6.1 Space and Time Overhead

A concern with cryptographic hashing is the amount of overhead that is incurred. In some cases, a provenance token may be larger than the input data records! Moreover, cryptographic hashing techniques are computationally intensive. Thus, we begin by looking at different space/security (and collision frequency) trade-offs.

Baseline performance. To establish a baseline for our measurements, we record the performance characteristics of the queries without provenance capture, in Table 1. Note the diversity in input sizes, output state size, and execution time. We do not view our PROVision platform as interchangeable with PostgreSQL, since it runs over unindexed external data and is written in Java as opposed to highly-memory-optimized C; nonetheless we include PostgreSQL running times for TPC-H queries to calibrate our performance. We are unaware of a system that can run our other queries' UDFs and trace their provenance.

Table 1: Baseline workflow, no provenance computation.

	input size	outputs	exec time	PostgreSQL load+exec
TPC-H Q1	71MB	735B	5.2s	2.7+1.2s
TPC-H Q3	90.4MB	747B	4.9s	3.2 + 0.18s
TPC-H Q5	90.5MB	258B	7.4s	3.3 + 0.14s
Genome	3.5GB	127GB	13.4hr	-
Magellan	615KB	615KB	4.3min	-
DuDe	4.6MB	116GB	25.3min	-

This table sets the baseline against which we will later plot normalized execution times and space overhead. (Space will be normalized against input, output, and stored intermediate result sizes.) We consider several additional baselines for provenance capture and storage. The *ProvCSV* method captures provenance in memory as polynomial expressions, and ultimately writes them alongside the output in a CSV file [32]. The Perm system [16] extends PostgreSQL with (non-compressed, non-tamper-resistant) provenance, for queries that run in PostgreSQL. Finally we adapted the provenance compression techniques of Chen et al. [10], in consultation with the authors, to work with PROVision's query model.

6.1.1 Hashing Schemes vs Space. Figures 4 and 5 show the space overheads incurred in tracking and storing provenance, for our various workloads. The TPC-H queries include joins and simple aggregation, whereas the scientific and ETL tasks involve substring extraction, approximate match, and ranked, thresholded computations. These sizes are *without* physical-level data compression. As noted, we normalize against *Baseline* space overhead; and against *ProvCSV*, which computes provenance in-memory, and stores provenance expressions as strings. Where feasible, we also compare against *Perm* (for TPC-H); and Chen et al. [10]'s compression strategy, re-implemented for our query model in consultation with the authors, which we term *NetworkProv*.

PROVision stores and cryptographically certifies the entire query expression. In contrast, *Perm* only stores the input "witnesses" responsible for an output tuple; and the *NetworkProv* approach stores inputs and output while eliminating intermediate nodes. A natural question is how our encoding compares to the alternatives. From the figures, we see that *Perm* and *NetworkProv* exhibit similar characteristics, with little overhead for TPC-H Q3 and Q5 (which have small result sizes) and fairly substantial overhead for TPC-H Q1 (which has fairly many witnesses of each output due to aggregation). For the ETL tasks, *NetworkProv* shows overhead similar to *PROVision-64* (slightly worse for DuDe and slightly better for Genome). Note that both *Perm* and *NetworkProv* will need to do more computation at query time to produce intermediate results.

To get insights into the overhead of tamper resistance vs. storage scheme, we compare different settings for PROVision, trading between token size (which affects I/O and even computation costs) and collisions and probabilistic security guarantees: we explore full cryptographic hashing and compare with options that fit within the storage system's highly efficient **long** and **int** datatypes. The cryptographically secure method (the *PROVision-F* bar, which uses full 512b Blake2 hashes) adds 4-50% overhead versus prior approaches like *NetworkProv*. For applications that need strong security guarantees, this overhead, while nontrivial, is certainly not prohibitive.

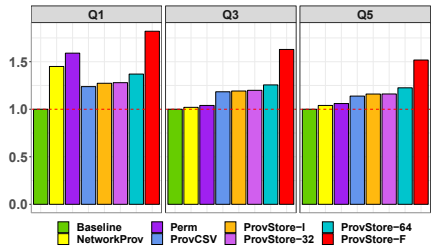


Figure 4: Space overhead: TPC-H queries.

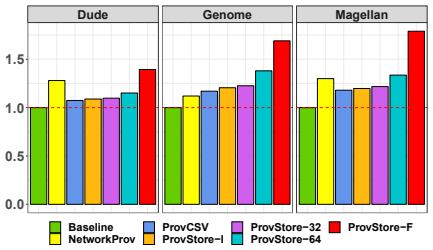


Figure 5: Space overhead: ETL tasks.

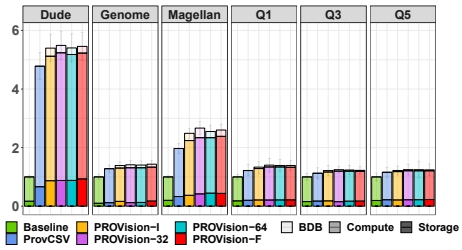


Figure 6: Time overhead for provenance.

Table 2: Theoretical tamper resistance vs space overhead.

Sys. Setting	Nbr. Ops.	Space Overhead
PROVision-F	2^{256}	40% to 90%
PROVision-64	2^{32}	15% to 39%
PROVision-32	2^{16}	10% to 20%

Table 3: Actual collisions observed in workloads.

PROVision	Q1	Q3	Q5	Genome	Magellan	DuDe
PROVision-I	115	123	129	11.4K	1245	241K
PROVision-32	104	116	119	10.5K	1223	238K
PROVision-64	0	0	0	0	0	0
PROVision-F	0	0	0	0	0	0

In some settings, e.g., if the source data is cryptographic signed, users may be able to tolerate reduced tamper resistance in the provenance structure — so we consider several alternatives. The *PROVision-I* configuration uses the computationally efficient Java `String hashCode()`, which is fairly weak. *PROVision-32* uses the same space, but replaces the standard hash function with a 32b version of the Blake2 cryptographic hash. Finally, *PROVision-64* uses a larger, 64b version of the Blake2 hash, which should have much better collision resistance as well as limited tamper resistance. We see that *PROVision-32* is preferable to *PROVision-I*, because it has fewer collisions with the same space overhead (10–20%). A reasonable trade-off that has high collision resistance and *some* tamper resistance is *PROVision-64*, with 15–39% space overhead. However, testing for tampering here requires access to a signed copy of the original database, since collisions among hashes in the Merkle tree can be fairly easily generated by an adversary.

Table 2 shows the theoretical trade-off between the security level and the space efficiency: "Nbr. Ops" represents the number of hashing operations before we expect a collision. In expectation, an adversary needs this many operations to find a value that could be substituted for the actual input. In principle, *PROVision-F* needs 2^{256} hash attempts to find a collision. For our actual datasets and query workloads, we did not observe any collisions with 64-bit or higher hashing, as shown in Table 3. For 32-bit hashing, we see that the Java hash function (PROVision-I) is actually reasonably close to the 32-bit cryptographic hash (PROVision-32) in collision resistance, even if it does not provide the same theoretical guarantees.

Sources of provenance overhead. PROVision requires space proportional to a query's intermediate state: each intermediate tuple results in a node in the graph; each provenance expression is stored in multiple *PTable* rows (*n* rows for an *n*ary operator). TPC-H Q1 (aggregation over a single input table) and the Magellan query — which compute many intermediate results that are aggregated or selected through top-*k* computations — have the highest overhead relative to their input + output size. The other queries tend to prune more of the input data and produce proportionally larger output results, so their relative overhead is less. In later experiments, we will show that these intermediate results often can be efficiently shared if they appear repeatedly across queries.

Physical-level data compression. Our discussion has focused on the logical-level data encoding. PROVision also leverages PostgreSQL's CStore_FDW columnar storage extension and its physical-level data compression techniques. This provides a substantial compression ratio, of approximately 2.5-3.5 times, for all encodings.

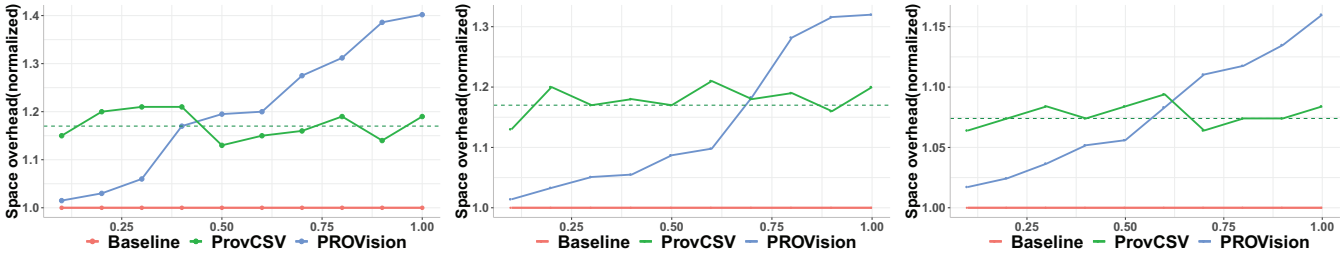
6.1.2 Hashing Schemes vs Execution Time. We consider the execution-time overhead of query computation, provenance graph I/O, and token management I/O, for the different configurations described in the previous section. In all cases our optimizer produces the same query plan. Figure 6 breaks down query costs into three components: provenance storage I/O (dark lower portions of the bars), query computation (longer, middle segment of the bar), and token management I/O (top of the bar). Provenance storage I/O is a relatively small fraction of the overall running times, since PROVision extensively caches reads in the graph and buffers writes.

Naturally, the query execution overhead depends on the computations being performed within the queries. Significant overhead (approximately 5x versus the baseline) occurs for the DuDe workload: this query computes an internal Cartesian product, requiring many intermediate results and the computation of their provenance, before it prunes to the most promising duplicates. All other queries are notably more efficient, with most adding 10–30% overhead under the different hashing schemes. We observe that *PROVision-32* is slightly (1% to 3%) slower than the other hashing schemes. This is because it is both a fairly expensive hash function to compute, yet it suffers from many collisions. *PROVision-64* and *PROVision-F* are indistinguishable, suggesting that the internal Blake2 algorithm does the same amount of work independent of the output hash size. *PROVision-I* is slightly faster due to its low CPU overhead, despite suffering from many collisions (recall Table 3).

We observed that provenance token management added approximately 5% overhead to the total execution time.

6.2 Provenance Reuse

To evaluate PROVision's ability to *share* storage across common expressions, we simulate a series of computations. Each makes small



(a) Genome query. (b) Magellan query. (c) DuDe query.

Figure 7: Space overhead for multi-query workloads, plotted against ratio of unique values.

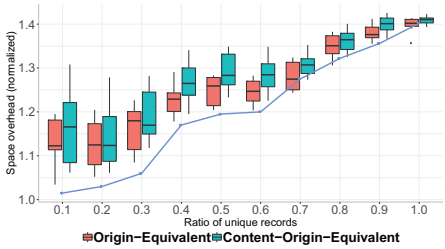


Figure 8: Space overhead vs. source equivalence scheme.

variations in its selection predicates and user-defined functions, over different input data files. We simulate the evolution of queries and datasets by varying: (1) the percentage of input records in common across different query executions; (2) the amount of overlap in the query plans being executed. We show the PROVision-64 configuration, which is representative of the other configurations.

Inputs with repeated sub-segments. To study the effects of repeated input data, for each scientific or ETL application we generate 10 queries $Q_1, \ldots, Q_{10}$ from the associated query template, and run each successive query over a different input set. For each application, we also generate 10 sets of input files, $D_1, \ldots, D_{10}$. Within these input sets, we control the number of repeated records (measured by the ratio of unique inputs to the total number of inputs). We ultimately run each Q_i over the corresponding D_i.

Figures 7(a)-(c) show, for our three respective query workloads, how the provenance space overhead changes as repetition occurs in the inputs. On the left side of the plot, duplicates are common (the ratio of unique input values is low); PROVision approaches the cost of the baseline, because provenance space is amortized across many repetitions. As we reduce duplication, PROVision dominates the *ProvCSV* approach until about 45-75% of the values are unique. The crossover point for Genome (which is aggregation-heavy) is lower than the other two tasks, which are more join-heavy. While our experiment studies repetition within the input file(s), the same "compression" benefits apply for queries executed across different inputs that share records.

Different source token equivalence schemes. The above experiments adopt a *value-equivalent* formulation to encode the source tokens: records are considered to match based on value-equivalence, independent of input file. Section 4.1.1 proposed several alternative definitions. To study how stricter definitions of equivalence affect provenance reuse, we simulate the evolution of data and queries over time as follows. We take 10,000 input sequences from the **Genome** dataset. We generate 10 files, each containing 1K samples from the input, with controlled overlap. We randomly assign a name to each file (potentially giving two files the same name, representing different versions of the file over time), and execute the Genome query. Here, the same record may occur in a file with a different name (making it no longer origin-equivalent) and/or a different hash signature (making it no longer content-equivalent).

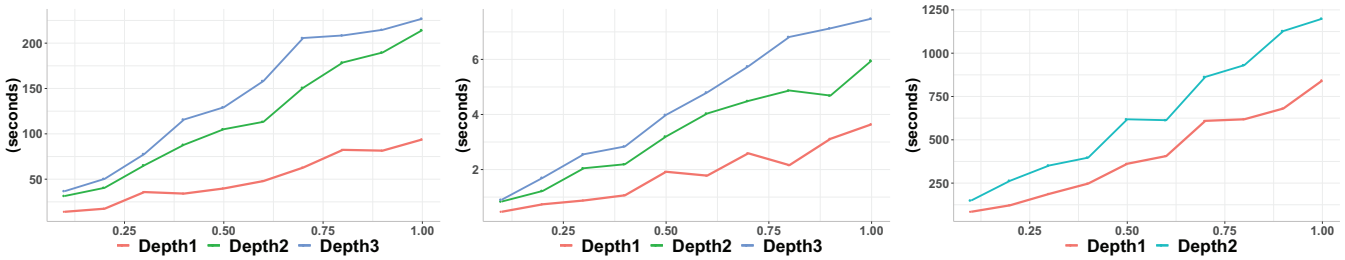
Figure 8 shows, over randomized runs, how our different equivalence definitions affect storage overhead, when compared to value-equivalence (the blue line). The x-axis shows the ratio of unique records (by value) across the workload, and the y-axis shows the space overhead versus the original data size. Note that when the ratio of unique records is low (left side of the plot), the storage overhead for COE and OE shows high variance and is considerably higher than for value equivalence. As the ratio of repeated records decreases, the schemes converge towards similar overhead.

Variations in queries across the workload. Our previous experiments repeated the same basic query (with different selection predicates) over different data — yielding the same query plan, and thus producing the same provenance structures over the same input data. To study the impact of changing queries, we define a very simple measure of query expression overlap. We consider two relational algebra operators to be identical if they have the same parameters, predicates, and user-defined function code. With this measure, we take a set of queries, and compute the ratio of *identical operators* as a proportion of the total number of operators; in essence this is a Jaccard similarity between two query expressions. Figure 10 plots the provenance space overhead (normalized against the setting where there are no common subexpressions) versus this ratio of identical operators, for the variations on our three basic queries. The figure clearly shows that as the number of shared operations increases, there are increasing efficiencies in storage as PROVision exploits common subexpressions. The aggregation-heavy Genome query benefits the most from sharing.

6.3 Provenance Retrieval and Comparison

We often need to *query* provenance — e.g., for reproducibility, to assess trust, or to compare different queries' results [21, 32].

Tracing to a fixed depth. We evaluate the provenance tracing times for different proportions of the overall output, in Figure 9



(a) Tracing Genome query output. (b) Tracing Magellan query output. (c) Tracing DuDe query output.

Figure 9: Provenance tracing times vs proportion of query output, to different depths.

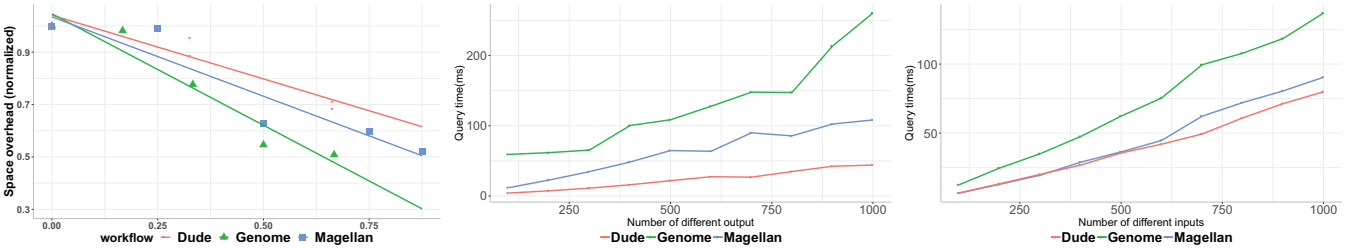


Figure 10: Space vs. Jaccard distance. **Figure 11: Divergent derivations query.** **Figure 12: Transitive closure query.**

(a)-(c). For each query, we plot tracing time against the fraction of the output traced. Each line represents the cumulative time to reach a given tracing depth (1 – 3 hops, which was the maximum depth of our queries). Tracing times are roughly linear in the size of the selected output, although for the Genome and Magellan queries at Depth 3, execution times slightly tail off at about 75% of the output size. These queries include an aggregation step that has high fan-in.

Divergent derivations. Scientists may need to compare slightly different queries, to see where their results are identical and where they diverge [32]. This involves tracing from the two queries' outputs, determining which paths trace back to common provenance derivations, and which do not (divergent derivations).We create a frontier set with the selected output nodes. Iteratively, we copy (and remove from the frontier) any nodes that are shared between the two query results: this can be tested by looking at expression equivalence, generally just by comparing token hash values. Such values represent *common* derivations. We then trace from the remaining nodes on the frontier to their inputs, and repeat the process. Each node represents a derivation that is unique to one of the queries, and is included as a *divergent* derivation. Figure 11 shows that DuDe and Magellan run in time approximately linear in the size of the outputs being traced, whereas Genome's times increase at a significantly higher rate due to large amounts of tuple grouping.

Transitive closure. A third type of provenance query traces back to inputs via transitive closure — to determine if output records are based on trustworthy inputs, or if they make use of certain records. Our last experiment randomly selects different numbers of inputs as starting nodes, and then computes their transitive closure in the provenance graph. Figure 12 shows that the query times are mostly linear in the size of the given inputs. Query time depends on the number of edges in the transitive closure, i.e., the number of rows retrieved in the PTable (Table 4). Consistent with our prior query experiments, Genome has the highest rate of increase due to its higher fan-in, whereas DuDe and Magellan are fairly similar.

Table 4: Look-ups/edges in the transitive closure.

Workflows	Depth	Num. lookups
Genome	8	6834
Magellan	6	5071
Dude	3	3500

7 CONCLUSIONS AND FUTURE WORK

In this paper, we developed a strategy for encoding fine-grained (semiring) provenance, such that repeated computations would be stored only once (derivation-based compression) and that provenance would be tamper-resistant. Cryptographic hashing provides a mechanism for quickly determining if a subexpression has been tampered with, and allows us to compare whether two subexpressions are identical. We developed a storage scheme for this model, including for collision resolution. Using our PROVision system, we studied the trade-offs between probabilistic security guarantees and performance. For future work, we would like to explore integration with GitHub and dependency managers for tracking binary and code versions; and to explore whether we might store provenance "templates" instead of full trees, to capture patterns more compactly.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was funded in part by NSF grants III-1910108, ACI-1547360, CCF 1763514, and NIH grant 1U01EB020954.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *Proc. VLDB Endow.* 5, 4 (2011), 346–357. https://doi.org/10.14778/2095686.2095693

[3] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, Maurizio Lenzerini and Thomas Schwentick (Eds.). ACM, 153–164. https://doi.org/10.1145/1989284.1989302

[4] Manish Kumar Anand, Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. 2009. Efficient provenance storage over nested data collections. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings (ACM International Conference Proceeding Series)*, Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold (Eds.), Vol. 360. ACM, 958–969. https://doi.org/10.1145/1516360.1516470

[5] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. BLAKE2: simpler, smaller, fast as MD5. Available from https://blake2.net/blake2.pdf.

[6] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Wasim Sadiq. 2012. Efficient provenance storage for relational queries. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, Xue-wen Chen, Guy Lebanon, Haixun Wang, and Mohammed J. Zaki (Eds.). ACM, 1352–1361. https://doi.org/10.1145/2396761.2398439

[7] Louis Bavoil, Steven P. Callahan, Carlos Eduardo Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. 2005. VisTrails: Enabling Interactive Multiple-View Visualizations. In *16th IEEE Visualization Conference, IEEE Vis 2005, Minneapolis, MN, USA, October 23-28, 2005, Proceedings*. IEEE Computer Society, 135–142. https://doi.org/10.1109/VISUAL.2005.1532788

[8] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. 2013. PROV-DM: The PROV Data Model. https://www.w3.org/TR/prov-dm/

[9] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 993–1006. https://doi.org/10.1145/1376616.1376715

[10] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. 2017. Distributed Provenance Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 203–218. https://doi.org/10.1145/3035918.3035926

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

[12] Laura Chiticariu and Wang Chiew Tan. 2006. Debugging Schema Mappings with Routes. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 79–90. http://dl.acm.org/citation.cfm?id=1164136

[13] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. 2013. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics* 29, 1 (2013), 15–21.

[14] Uwe Draisbach and Felix Naumann. 2010. DuDe: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB)*.

[15] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. 1999. Query Optimization in the Presence of Limited Access Patterns. In *15èmes Journées Bases de Données Avancées, BDA 1999, Bordeaux, 25 - 27 octobre 1999. (Informal Proceedings)*, Christine Collet (Ed.). Actes, 41–60.

[16] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 174–185. https://doi.org/10.1109/ICDE.2009.15

[17] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 209–218. https://doi.org/10.1109/ICDE.1993.344061

[18] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2007. Update Exchange with Mappings and Provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, 675–686. http://www.vldb.org/conf/2007/papers/research/p675-green.pdf Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[19] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, Leonid Libkin (Ed.). ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[20] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

[21] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2010. Querying data provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 951–962. https://doi.org/10.1145/1807167.1807269

[22] Pradap Konda, Sanjib Das, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, et al. 2016. Magellan: toward building entity matching management systems over data science stacks. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1581–1584.

[23] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2019. PUG: a framework and practical implementation for why and why-not provenance. *VLDB Journal* 28, 1 (2019), 47–71.

[24] Xueping Liang, Sachin Shetty, Deepak K. Tosh, Charles A. Kamhoua, Kevin A. Kwiat, and Laurent Njilla. 2017. ProvChain: A Blockchain-based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE Computer Society / ACM, 468–477. https://doi.org/10.1109/CCGRID.2017.8

[25] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurr. Comput. Pract. Exp.* 18, 10 (2006), 1039–1065. https://doi.org/10.1002/cpe.994

[26] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, Carl Pomerance (Ed.), Vol. 293. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[27] Ricardo Neisse, Gary Steri, and Igor Nai Fovino. 2017. A Blockchain-based Approach for Data Accountability and Provenance Tracking. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*. ACM, 14:1–14:10. https://doi.org/10.1145/3098954.3098958

[28] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. 2006. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1067–1100.

[29] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 2.

[30] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (2018), 719–732. https://doi.org/10.14778/3184470.3184475

[31] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment* 12, 9 (2019), 975–988.

[32] Nan Zheng, Abdussalam Alawini, and Zachary G. Ives. 2019. Fine-Grained Provenance for Matching & ETL. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 184–195. https://doi.org/10.1109/ICDE.2019.00025

[33] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 295–310. https://doi.org/10.1145/2043556.2043584