ProvCite: Provenance-based Data Citation

Yinjun Wu
University of Pennsylvania
wuyinjun@seas.upenn.edu

Abdussalam Alawini
University of Illinois at Urbana-Champaign
alawini@illinois.edu

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

Susan Davidson
University of Pennsylvania
susan@seas.upenn.edu

ABSTRACT

As research products expand to include structured datasets, the challenge arises of how to automatically generate citations to the results of arbitrary queries against such datasets. Previous work explored this problem in the context of *conjunctive* queries and views using a Rewriting-Based Model (RBM). However, an increasing number of scientific queries are *aggregate*, e.g. statistical summaries of the underlying data, for which the RBM cannot be easily extended. In this paper, we show how a Provenance-Based Model (PBM) can be leveraged to 1) generate citations to conjunctive as well as aggregate queries and views; 2) associate citations with individual result tuples to enable arbitrary subsets of the result set to be cited (*fine-grained citations*); and 3) be optimized to return citations in *acceptable time*. Our implementation of PBM in ProvCite shows that it not only handles a larger class of queries and views than RBM, but can outperform it when restricted to conjunctive views in some cases.

PVLDB Reference Format:
Yinjun Wu, Abdussalam Alawini, Daniel Deutch, Tova Milo and Susan Davidson. ProvCite: Provenance-based Data Citation. *PVLDB*, 12(7): 738-751, 2019.
DOI: https://doi.org/10.14778/3317315.3317317

1. INTRODUCTION

The notion of "research products" has expanded to include structured datasets, and there is growing interest within both the digital library and computer science communities to be able to cite information extracted by queries over these datasets. Citations play a significant role in giving credit to those responsible for the data, and enable the data to be later found or reproduced. Much like a citation to traditional research products such as journal or conference papers, a citation to the result of a query over a structured dataset should include snippets of information describing the dataset (analogous to a title), who is responsible for the dataset (e.g. the PI or contributors/curators of the data), as well as information about how to find the dataset (e.g. the http address, database version, and query).

Several computational challenges must be addressed in developing a data citation system [11]. First, since the number of possible queries over a database is very large, it is infeasible to associate a citation to each query. Instead, one should be able to specify citations for a small number of frequent queries and use them to automatically derive citations to other "general" queries. Second, this must be done with an acceptable time overhead, e.g. without adding significantly to the query response time. Third, it is useful to allow the user to select a subset of the query result for which a citation should be generated, which we call "fine-grained" citations. This need arises in many different scientific applications, in particular neuro-imaging [32].

In prior work, we proposed a general framework to *automatically generate fine-grained citations for general queries* [6, 40]. The approach is based on a model of *citation views* [11, 22, 21]: Frequently posed queries are defined as views with associated citations. A query against the database is rewritten in terms of these views, and the associated citations are used to construct a citation for *each tuple in the query result*. Since a query may be rewritten by *jointly* using more than one view, or there may be several *alternate* ways to rewrite a query, the database owner may specify how citations are jointly or alternatively combined through *policies*. The framework also allows for fine-grained citations: the citations for each tuple in the query result are then *combined* to create a final citation for the specified subset of the result, which is given as another policy. Policies give an interpretation for the joint, alternate and combined use operators, for example, taking the union, intersection or join of the citations. In the remainder of the paper, we will call the model used in [40] the *Rewriting-Based Model (RBM)* since it extends query rewriting using views algorithms to work at the tuple level.

A shortcoming of RBM, however, is that it addresses a limited class of queries – (non-recursive) conjunctive queries and conjunctive views – and cannot be used in applications in which the queries and views involve aggregates (such as SUM, MIN, AVG) or user-defined functions. However, there is a growing number of biomedical applications which extract *summaries* from databases by issuing aggregate queries, in which views involve aggregation. In these cases, the techniques of [40] cannot be used.

One such example is Hetionet, a database that "encodes" biology by integrating various types of biological informa-

Proceedings of the VLDB Endowment, Vol. 12, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3317315.3317317

tion from different publicly available resources [30, 31]. As data is copied from these source datasets, citation information (generally in the form of traditional publication IDs) is also copied and should be propagated to the results of queries. The majority of queries against this database involve aggregation to retrieve statistical information.

Another example, which requires both aggregate queries and aggregate views, is GENCODE [29], an encyclopedia of genes and gene variants whose goal is to identify all functional elements in the human genome using annotations. The gene annotation process involves a combination of automatic annotation, manual annotation, and experimental validation. For genes that are manually annotated, information is maintained about the responsible research groups. Statistics are also provided for every gene – an *aggregate view* over the genes – which has another type of citation giving credit to the creators of the aggregate view. Common queries over GENCODE also involve aggregation. For instance, one query computes statistics for every *type* of gene.

In this paper, we address the problem of automatically generating fine-grained citations when both the queries and views may involve aggregates. Although at first glance it would appear that rewriting techniques for aggregate queries [42, 39, 25, 17, 16] could be used, these techniques reason at the schema level for the *entire query result* rather than at the level of individual tuples, which is required for fine-grained citations. Extending the implementation in [40] to use ideas from query rewriting for aggregate queries is possible when views are conjunctive, but is problematic when views involve aggregation since aggregation blurs the connection between tuples in the input relations and tuples in the result.

Instead, to support aggregation, we use the observation pointed out in [11, 5] that there is a strong connection between data provenance and data citation – and the provenance of aggregate queries is well understood. We therefore adopt a *Provenance-Based Model (PBM)* that captures the connections between a result tuple and tuple(s) in views.

Example. Recall that GENCODE is an encyclopedia of information about genes and gene variants. Suppose that one of the views defined by the DBA is V_{gene}, which counts the number of genes for each gene type, but only retains the gene types (groups) with more than 10 genes. This corresponds to an aggregate query with a HAVING-clause in SQL. V_{gene} has an associated citation query which describes the citation for each tuple in the view.

Now suppose that a query Q counts the number of genes *whose gene ids are smaller than 50* for every gene type. Then some tuples in the query result will appear in V_{gene}, and therefore carry the associated citation; these are the gene types with more than 10 genes and in which all gene ids are smaller than 50. Other tuples in the query result may not appear in V_{gene}, and would not therefore carry the citation associated with V_{gene}; these are the gene types which include some genes with ids 50 or greater or include fewer than 10 genes.

Traditional query rewriting using views techniques would therefore conclude that V_{gene} is *not useful* for Q. Furthermore, the RBM tuple-level techniques proposed in [40] could not detect whether V_{gene} is useful for a given tuple in the result of Q. However reasoning over the *provenance* of result and view tuples could determine this, and return the citation for V_{gene} for result tuples as appropriate.

Approach. We develop a citation system called ProvCite, which executes over a provenance-enabled relational database system. As in [40], the DBA defines the citation views and policies to be used. When a query is submitted, all potential *view mappings* are computed, which represent how views can potentially rewrite this query. The decision of which views are valid, however, depends on the particular result tuple, and for this the provenance of the result tuple is compared with the provenance of view tuples. While the user is presented with the query result and examines it to determine the subset of interest, *covering sets* are calculated from the valid views for every result tuple, representing alternate rewritings in which sets of views are jointly used. When the result subset is selected, the citation for the selected query subset can be immediately generated.

Our initial fear in developing ProvCite was that, although the approach is interesting since it develops a novel connection between citation and provenance, it would be unacceptably slow since provenance expressions are typically very large. *Surprisingly, the results of this paper not only show that PBM is feasible and extends results in [40] to aggregate queries and views, but that our optimized computation allows it to even outperform our previous RBM approach in some cases.*

Contributions of this paper include:

1. A framework formalizing the connection between data provenance and data citation.
2. A semantics for generating citations to the results of aggregate queries with *general aggregate functions* given a set of either aggregate or conjunctive views using provenance in the view and query instances.
3. An implementation of PBM called ProvCite, which automatically generates fine-grained citations for the results of general queries, where both the queries and views may involve aggregates. The implementation includes multiple dedicated optimizations.
4. An experimental study using both *synthetic* and *realistic* workloads, showing the efficiency of the approach and the effectiveness of the developed optimizations.

The rest of this paper is organized as follows. Related work is discussed in Section 2, and the running example and preliminaries are given in Section 3. Details of the PBM and its implementation in ProvCite are presented in Sections 4 and 5 respectively. Section 6 gives experimental results before concluding in Section 7.

2. RELATED WORK

Data citation. Principles for data citation have been proposed within the digital library community[1, 24] and include: 1) identification and access to the cited data; 2) persistence of the cited data; and 3) completeness of the reference [34, 38, 10, 2]. The community also recognized the importance of citations to aggregate data [1], as have various scientific communities [29, 31, 35]. More recently, data citation has captured the attention of database researchers, who formulated computational challenges [11, 21]. To address these challenges, a model of *citation views* was defined in [22] and implemented in [6, 40]. However, this work was limited to conjunctive queries and views without addressing aggregates.

Query rewriting using views. Data citation is closely related to the problem of query rewriting using views. Rewriting relies on notions of *containment* and *equivalence* of queries [28], and has been extensively studied in the context of conjunctive queries [13, 15, 36, 3] as well as aggregate queries [18, 19]. Various algorithms have been designed to rewrite aggregate queries. For example, [39, 25] provide algorithms for determining whether a materialized view is usable for answering an aggregate query by considering both conjunctive and aggregate views. In [42], an algorithm is given to handle nested subqueries and multidimensional aggregations in queries and views. However, only standard aggregate functions (e.g. SUM, COUNT) are considered in [42, 39, 25]; general aggregate functions (such as user defined aggregate functions) cannot be used. The problem of general aggregate functions is considered in [17], and [16] bridges the gap between theory and practice by providing implementation suggestions. However, to our knowledge, there is no work which considers how to rewrite queries using *general aggregate views with having clauses*.

Data Provenance. Data provenance identifies where a piece of data came from and the process by which it arrived in the database [12]. It has been used to track the dependencies between inputs and outputs, detect errors in complex workloads, and provide explanations for debugging purposes. Various formulations of provenance have been studied, such as *why- and where-provenance* [12], *why-not-provenance* [14], and the *provenance semirings* framework [27] (extended in e.g. [8, 41]). This framework has been used to implement several practical provenance-enabled database systems, such as ORCHESTRA [33] and GProM [9]. The connection between data citation and provenance was discussed in [11] and explored but not formalized in [5]. This paper develops those ideas further, provides an implementation based on a provenance-enabled database system, and shows the feasibility of the approach.

3. PRELIMINARIES

In this section, we introduce the running example, review the notions of *citation views* [22], *view mapping* and *validity* of view mappings [40], and then show why the RBM of [40] cannot be extended for aggregate views, motivating the need for *provenance*.

3.1 Running example: GENCODE

We use a simplified schema from GENCODE as our running example. In this database, information is structured hierarchically: each gene is associated with one or more transcripts, and each transcript has one or more exons. Genes, transcripts, and exons may all be annotated with tags, which are created either by human experts or by programs. A simplified schema is as follows:

Gene(GID, Name, Type)
Gene2tag(GID, annot) GID references Gene
Transcript(TID, Name, Type, GID) GID references Gene
Transcript2tag(TID, annot) TID references Transcript
Exon(EID, Level, TID), TID references Transcript
Exon2tag(EID, annot), EID references Exon

Relations Gene2tag, Transcript2tag and Exon2tag capture the annotations (*annot*) for relation Gene, Transcript and Exon respectively.

Citation views [22] define views of the database to which citations have been specified. A citation view consists of a *view query*, defining the subset of the data to which the citation is attached; a *citation query*, which retrieves required citation information; and a *citation function*, which formats the information retrieved by the citation query to provide the final citation, e.g. in JSON, BibTex or RIS format.

Below we show several views for GENCODE, which are expressed using S-Datalog [20], an extended version of Datalog that allows aggregates:

$$
\begin{array}{ll}
\lambda G.V_1(G, Ty) & :-Gene(G, N, Ty), G \leq 2 \\
V_2(Ty, COUNT(G)) & :-Gene(G, N, Ty), Ty = \text{`}rRNA\text{'} \\
V_3(T1, E, G1, L) & :-Transcript(T1, N1, Ty1, G1), \\
 & \quad Exon(E, L, T2), T1 = T2, E \leq 2 \\
V_4(G1, COUNT(T1)) & :-Transcript(T1, N1, Ty1, G1), \\
 & \quad Exon(E, L, T2), T1 = T2, L \leq 2 \\
V_5(G1, MAX(L), COUNT(E)) & \\
 & :-Transcript(T1, N1, Ty1, G1), \\
 & \quad Exon(E, L, T2), T1 = T2 \\
V_6(G, COUNT(T)) & :-Transcript(T, N, Ty, G), T \leq 2
\end{array}
$$

V_1 and V_3 are simple conjunctive queries. V_1 is *parameterized* by the gene id G, meaning that it defines a family of views, one for each gene. Each view in this family consists of a single tuple. In this way, each gene may have different citation, giving credit to the person or program who annotated that gene. In contrast, V_3 is not parameterized, which indicates that the same citation is shared across all the view tuples. The other four views are aggregate views. Their meaning is: for each binding of variables in the body, group over the variables in the head (called *grouping variables*) and apply the aggregate(s) to each group. Each aggregate function along with its arguments is called an *aggregate term*, in which the arguments are called *aggregate variables*. Thus V_2 could be translated into SQL as:

```
SELECT G.Type, COUNT(G.GID)
FROM Gene G
WHERE G.Type = 'rRNA'
GROUP BY G.Type
```

which counts the number of 'rRNA' genes. Similarly, V_4 counts the number of transcripts per gene which are associated with one exon whose level is no more than 2. V_5 returns the maximal level of exons and the count of exons for each gene. $V6$ counts the number of transcripts for each gene whose transcript ids are no more than 2.

3.2 Query rewriting: View mappings

Given a (general) query Q and a set of views $\mathcal{V}$, the RBM implementation in [40] starts by building a set of *view mappings* $\mathcal{M}$ from $\mathcal{V}$ to Q. For each query tuple, RBM then reasons about the *validity* of each view mapping in $\mathcal{M}$ and constructs *covering sets*. Each covering set is a *maximal*, *non-redundant* set of *valid* view mappings, i.e. sets for which no other view mappings can be added to *cover* more subgoals and head variables in Q, nor can any be removed and still cover the same subgoals and head variables in Q. The citations associated with views in the covering sets are later used to construct the final citations.

A *view mapping* M consists of a *relation mapping*, h and *variable mapping*, ϕ (denoted $M = (h, \phi)$). The former, h, maps relational subgoals in a view V to relational subgoals

with the same relation names in query Q, while the latter variable mapping ϕ is an induced mapping by h.

Intuitively, for a query tuple t, view mapping M is valid iff some portion of t *appear* in the instance of V under M. The reasoning depends on examining the lambda variables, the head variables and the predicates of views.

Example 1. Consider the following Conjunctive Query:

$Q_1(T, Gid)$: $-Gene(Gid, Name, T), Gid \leq 3$

There is a view mapping $M_{11} = (h_{11}, \phi_{11})$ from V_1 to Q_1, in which the relation mapping is h_{11} is $\{Gene(G, N, Ty) \rightarrow Gene(Gid, Name, T)\}$ while the variable mapping ϕ_{11} is $\{G \rightarrow Gid, N \rightarrow Name, Ty \rightarrow T\}$. M_{11} is only valid for query tuples with $Gid \leq 2$, since those query tuples can appear in the instance of $V_1(D)$ by satisfying predicate ($G \leq 2$); the head variables G (which is also a lambda variable) and Ty in V_1 are mapped to head variables Gid and T in Q_1. RBM determines validity by checking whether the portion of a query tuple t "exists" under the view mapping in the view instance, which is achieved by evaluating the satisfiability of view predicates in the query instance. In this example, the instance of Q_1 along with the evaluation of the predicate ($Gid \leq 2$) (derived from the predicate of V_1) for each query tuple is shown in Table 6 (ignore the last column for now). The truth value indicates whether or not the query tuple exists in the view instance under the view mapping. For instance, $t_{q_1 1}$ satisfies all the predicates in V_1, i.e. ($Gid \leq 2$) since $t_{q_1 1}$ "exists" in the instance of $V_1(D)$ (i.e. $t_{v_1 1}$, under the mapping M_{11}); in contrast, $t_{q_1 3}$ is missing from $V_1(D)$.

For query tuples such that M_{11} is a valid view mapping, a covering set can be constructed: $\{M_{11}\}$. After the user selects the query tuples of interest, the *citation queries* associated with V_1 are executed and the *citation function* is applied to construct formatted citations. (see the last but one column in Table 6 for the citations for each query tuple.)

3.3 The need for provenance

We next illustrate why RBM fails for aggregate views, motivating the need for *provenance*.

Example 2. Consider the following query which is constructed by adding one aggregate function to Q_1:

$Q_2(T, COUNT(Gid)) : -Gene(Gid, Name, T), Gid <= 3$

Note that only V_1 and V_2 can provide candidate view mappings $M_{12}(= (h_{12}, \phi_{12}))$ and $M_{22}(= (h_{22}, \phi_{22}))$ for Q_2 since they include the same relational subgoals in the body. M_{12} and M_{22} have the same form, i.e,
$h_{12}(h_{22}) = Gene(G, N, Ty) \rightarrow Gene(Gid, Name, T)$ and $\phi_{12}(\phi_{22}) = \{G \rightarrow Gid, N \rightarrow Name, Ty \rightarrow T\}$. Under the two mappings, neither V_1 nor V_2 can be used to rewrite Q_2 for an arbitrary database instance since V_1 has one logically stronger predicate than Q_2, and we can find an instance D for V_2 such that the first three genes are not 'rRNA'. In this case, V_2 and Q_2 aggregate over different set of tuples from the *Gene* relation.

Still, *some* query tuples may be computed using view tuples of V_1 and V_2 given a database instance D, which captures the concept of *fine-grained citation* proposed in [40]. To illustrate, an instance D is given in Tables 1-3, and the resulting instances of V_1, V_2 and Q_2 are shown in Tables 4, 5 and 7 respectively (ignore the last column for now). Note

Table 1: Instance of relation *Exon* with provenance

	EID	Level	TID	prov
t_{e1}	1	1	1	e_1
t_{e2}	2	3	2	e_2
t_{e3}	3	2	2	e_3
t_{e4}	4	2	2	e_4

Table 2: Instance of relation *Gene* with provenance

	GID	Name	Type	prov
t_{g1}	1	TF	TEC	g_1
t_{g2}	2	FH	rRNA	g_2
t_{g3}	3	RP1	rRNA	g_3
t_{g4}	4	IYD	rRNA	g_4
t_{g5}	5	EPN	mRNA	g_5

Table 3: Instance of relation *Transcript* with provenance

	TID	Name	Type	GID	prov
t_{t1}	1	MB-203	TEC	1	r_1
t_{t2}	2	PC-203	rRNA	2	r_2
t_{t3}	4	HP-218	rRNA	2	r_3

that the citations for each view tuple are also included in the view instances (the column with the green background).

To follow the idea of checking the *existence* of a query tuple in the view instance, a plausible approach is to extend RBM to aggregate queries by employing a special built-in function *agg* (the "array_agg" function in PostgreSQL) to collect the evaluations of the view predicates for every query tuple before aggregation is applied. For example, the instance of Q_2 is shown in Table 7 along with the evaluation results of view predicates collected by *agg* (see the third and fourth columns). Note that, since two tuples are generated before the aggregate function is applied for tuple $t_{q_2 2}$ (i.e. $t_{q_1 2}$ and $t_{q_1 3}$ from Table 6), the evaluation of $agg((Gid \leq 2))$ will result in a *set* of boolean values of size 2, which, intuitively, is derived by aggregating over $t_{q_1 2}$ and $t_{q_1 3}$ in Table 6. The existence of "false" indicates that before aggregation, one query tuple (i.e. $t_{q_1 3}$) was missing from the view instance $V_1(D)$, thus V_1 is not valid for $t_{q_2 2}$. In contrast, for $t_{q_2 1}$ the evaluation of $agg((Gid \leq 2))$ is only "true", meaning that V_1 is valid.

However, simply checking the existence of query tuples in the view instance for aggregate views is not enough. For example, for query tuple $t_{q_2 2}$, although the evaluations of the predicate $Ty =$ '$rRNA$' (from V_2) does not include "false", indicating that all the query tuples before aggregation exist in the view instance, the aggregate result of $t_{q_2 2}$ (i.e. 2) does not match that of $t_{v_2 1}$ (i.e. 3). The reason is that three tuples $t_{g2} - t_{g4}$ from relation *Gene* are used to construct $t_{v_2 1}$, while $t_{q_2 2}$ is derived from only two of them ($t_{g2} - t_{g3}$). This means that the reasoning should not only capture the *existence* of query tuples in the view instance but also the *exact matching* of the aggregate results between query tuples and view tuples. We therefore adopt an alternative model, called the *Provenance-Based Model* (*PBM*), which uses *provenance*.

4. PROVENANCE-BASED MODEL

We now present the model for determining valid view mappings for each query tuple using provenance.

4.1 Basic concepts

We start by introducing the basic concepts that will be used in our model.

Granularity of queries and views. An essential step in determining the validity of a view mapping $M = (h, \phi)$ is to compare the schemas of Q and V, and detect whether V keeps all necessary variables in its head. In particular, if the aggregate view V has the set of grouping variables $\{Y_1, Y_2, \ldots, Y_m\}$, then $\{\phi(Y_1), \phi(Y_2), \ldots, \phi(Y_m)\}$ should be a superset of the grouping variables of Q, $\{X_1, X_2, \ldots, X_k\}$. If $\{\phi(Y_1), \phi(Y_2), \ldots, \phi(Y_m)\} = \{X_1, X_2, \ldots, X_k\}$, we say Q has the *same granularity* as V.
Otherwise, if $\{\phi(Y_1), \phi(Y_2), \ldots, \phi(Y_m)\} \supsetneq \{X_1, X_2, \ldots, X_k\}$, we say V has *finer granularity* than Q.

How-provenance. We use the notion of *how-provenance* introduced in [27]. It starts by annotating each base relation tuple with a unique *provenance token*, and propagates those tokens along with the tuples to the query result. Each tuple in the query result then has a *how-provenance polynomial* expressed using + (alternate use) and ∗ (joint use) to indicate *how* base relation tuples contribute to the query result. Each *how-provenance polynomial* is composed of multiple *how-provenance monomials* which are joint-use terms expressed with ∗. For example, the provenance polynomial for tuple $t_{q_2 2}$ in $Q_2(D)$ is $g_2 + g_3$, which has two *how-provenance monomials* (see Table 7). It means that two tuples from relation *Gene* with how-provenance tokens g_2 and g_3, respectively were used to create $t_{q_2 2}$. The ∗ operator is used if there are multiple relational subgoals in the query body, in which case base tuples are *jointly used* to create the result.

Table 4: $V_1(D)$ with how-provenance and citation

	G	Ty	citation	prov
$t_{v_1 1}$	1	TEC	{Group: ['Joe']}	g_1
$t_{v_1 2}$	2	rRNA	{Group: ['Liu']}	g_2

Table 5: $V_2(D)$ with how-provenance and citation

	Ty	COUNT(G)	citation	prov
$t_{v_2 1}$	rRNA	3	{Group: ['Lee']}	$g_2 + g_3 + g_4$

Table 6: $Q_1(D)$ with how-provenance

	T	Gid	$(G \leq 2)$	citation	prov
$t_{q_1 1}$	TEC	1	T	Group:['Joe']	g_1
$t_{q_1 2}$	rRNA	2	T	Group:['Liu']	g_2
$t_{q_1 3}$	rRNA	3	F		g_3

Table 7: $Q_2(D)$ with how-provenance

	T	COUNT (Gid)	agg $((G \leq 2))$	$agg((Ty =$ '$rRNA$'))	prov
$t_{q_2 1}$	TEC	1	{T}	{F}	g_1
$t_{q_2 2}$	rRNA	2	{T, F}	{T, T}	$g_2 + g_3$

Table 8: $Q_3(D)$ with how-provenance

	G	G'	prov
$t_{q_3 1}$	2	2	$r_2 * r_2 + r_2 * r_3 + r_3 * r_2 + r_3 * r_3$

To simplify reasoning over how-provenance polynomials, [7] defines a normal form as follows: first, the provenance tokens in each how-provenance monomial preserves the same order as the relational subgoals in the query body. Second, the exponent of every provenance token is forced to be 1. Third, the coefficient of every monomial in a how-provenance polynomial is forced to be 1 by breaking the monomials with coefficient greater than 1 into multiple how-provenance monomials, each corresponding to an *assignment* of the query atoms to database tuples.

Example 3. Consider the following query:

$$Q_3(G, G') : -Transcript(T, N, Ty, G), T >= 2, Ty = Ty', \\ Transcript(T', N', Ty', G'), T' >= 2$$

The provenance-aware query result is shown in Table 8 using the instance of *Transcript* in Table 3. Note that the second and the third monomials for the tuple $t_{q_3 1}$ are equivalent but correspond to different assignments, hence written differently ($r_2 * r_3$ vs $r_3 * r_2$). Further, the first how-provenance monomial of $t_{q_3 1}$ is written as $r_2 * r_2$ instead of the compact form (r_2^2). Furthermore, the coefficient of all the monomials in the how-provenance polynomial of $t_{q_3 1}$ is 1 rather than grouping them together.

For a query and a result tuple, [7] defines an *isomorphism* between *assignments* and the how-provenance monomials in a query. Borrowing these ideas, we define an isomorphism between relational subgoals and how-provenance monomials under an assignment γ, which relies on the normal form of how-provenance monomials mentioned previously.

Definition 1. **Isomorphism between how-provenance monomials and subgoals.** Given a conjunctive or aggregate query Q with relational subgoals $B_1, B_2, \ldots, B_m$, under an assignment γ, base relation tuples $t_{b1}, t_{b2}, \ldots, t_{bm}$ are assigned to relational subgoals $B_1, B_2, \ldots, B_m$ respectively to generate an output tuple, which can be written as $\gamma(B_i) = t_{bi} (i = 1, 2, \ldots, m)$ [8]. If tuple t_{bi} is associated with how-provenance token h_{bi}, then we say that under the assignment γ there is an *isomorphism* F between each relational subgoal B_i and each provenance token h_{bi} (call *isomorphism under an assignment* for short thereafter), which can be written as: $F(B_i|\gamma) = h_{bi}$ and $F^{-1}(h_{bi}|\gamma) = B_i$.

Returning to Example 3, consider the second provenance monomial $r_2 * r_3$ and corresponding assignment γ in query tuple $t_{q_3 1}$ of Table 8. Since t_{t2} and t_{t3} are associated with how-provenance tokens r_2 and r_3 respectively, there should be an isomorphism F under γ such that $F(Transcript(T, N, Ty, G)|\gamma) = r_2$ while $F(Transcript(T', N', Ty', G')|\gamma) = r_3$.

4.2 Validity conditions without aggregation

The validity conditions of view mappings include *schema-level conditions* and a *tuple-level condition*, which are satisfied by a view mapping M iff it is valid for a given query tuple. The validity conditions guarantee the same result as the conditions in [40] (proof omitted).

Definition 2. **Schema-level conditions.** A view mapping M from a conjunctive view V to a conjunctive query Q should satisfy the following conditions at the schema level if it is valid for some query tuples:

1. There exists at least one head variable $y \in \bar{Y}$ in V such that $\phi(y)$ is a head variable in Q; and
2. All lambda variables in V are mapped to variables in the body of Q.

Now suppose that head variables $Y_1, Y_2, \ldots, Y_r$ from V are mapped to head variables $X_1, X_2, \ldots, X_r$ from Q, which implies that $\phi(Y_i) = X_i$ $(i = 1, 2, \ldots, r)$. Then we say that the head variables $X_i (i = 1, 2, \ldots, r)$ are *covered* under M. Plus, a relational subgoal of Q is *covered* by M iff it is involved in M.

Definition 3. **Tuple-level condition.** Let the how-provenance polynomial of $t_q \in Q(D)$ ($t_v \in V(D)$) include a how-provenance monomial W (W') with corresponding assignment γ (γ') and the isomorphism F (F') under γ (γ'.)

Given a tuple t_q and a view mapping $M = (h, \phi)$ satisfying the *schema-level conditions* above, if we can find a tuple t_v such that the following condition holds, then we say that M is *valid* for the how-provenance monomial W in t_q: For each relational subgoal A_i in the view body that is involved in the view mapping M and mapped to relational subgoal B_j in the query body under M, then $F(B_j|\gamma) = F'(A_i|\gamma')$.

Furthermore, we say that the how-provenance monomial W' of t_v is *mapped* to the how-provenance monomial W of t_q under view mapping M.

Example 4. Let us revisit Example 1. The instance of Q_1 is shown in Table 6. We can show that M_{11} is a valid view mapping for the query tuple $t_{q_1 1}$ and $t_{q_1 2}$: The *schema-level conditions* are satisfied because the head variable Ty and G (which is also the only lambda variable) in V_1 are mapped to the head variable T and Gid in Q_1 respectively.

The *tuple-level condition* also holds for the two query tuples. For example, for $t_{q_1 2}$ (and view tuple $t_{v_1 2}$), for its single monomial, the assignment and isomorphism under the assignment are γ (γ') and F (F') respectively. Since under the view mapping $M_{11} = (h_{11}, \phi_{11})$, $h_{11}(Gene(G, N, Ty)) = Gene(Gid, Name, T)$, and $F'(Gene(G, N, Ty)|\gamma') = g_2 = F(Gene(Gid, Name, T)|\gamma)$. So we say that M_{11} is a valid view mapping for the how-provenance monomial g_2 for query tuple $t_{q_1 2}$. We can also prove that M_{11} is a valid view mapping for how-provenance monomial g_1 in tuple $t_{q_1 1}$.

4.3 Validity conditions with aggregation

The validity conditions for view mappings are next extended to handle aggregate queries and views, using the following intuition: *for a query tuple t, if 1) a set of view tuples can be used to compute t by applying some aggregate function(s) and 2) the view tuples and t are constructed by the same multiset of tuples from the base relations (captured by provenance), then the citation information of those view tuples can be used to construct the citation of t.*

We start by introducing requirements on the aggregate functions before formalizing this intuition.

4.3.1 Aggregate function requirements

A view mapping M, which maps an aggregate view V to an aggregate query Q, is valid for a query tuple only if the aggregate functions of V and Q satisfy certain requirements; in particular, [16] formalizes the notion of a *well-formed aggregate function.* Loosely speaking, a well-formed aggregate function can be characterized by some initial "mapper" function, followed by a "reduce" function, followed by a "finalize" function, which we will call a *terminating function.* It is easy to see that some common aggregate functions such as SUM,MIN, MAX, COUNT and AVG are well formed.

For example, the "mapper" function for AVG takes a set of values, $\{d_1, \ldots, d_k\}$, and maps each number d_i to a pair $(d_i, 1)$. The result of the reduce function is still a pair whose first element represents the sum of the d_i's and second element represents the count (k). The "finalize" function divides the first element by the second element. Similarly, SUM maps each d_i to itself and takes the sum of all d_i's in the reduce step; "finalize" is the identity function.

Invertibility. A well-formed function is invertible iff its terminating function is invertible. For example, SUM is invertible whereas AVG is not. Invertibility is important for determining the validity of view mappings when the view has a finer granularity than the query, as illustrated below.

Example 5. Consider the following query:

$$Q_5(COUNT(G)) : -Gene(G, N, Ty), Ty = \text{`}rRNA\text{'}$$

By referencing Section 3.1, V_2 computes a coarser-grained aggregation result than Q_5 does. Both share the same aggregate function $COUNT$, which is invertible. This means that we can take the sum of the aggregation results in V_2 to get the result of Q_5 under the obvious view mapping M_5.

However, if we replace $COUNT$ with AVG for both Q_5 and V_2, the aggregation result in V_2 will not be useful to compute the aggregation result in Q_5 under M_5; the intermediate sum and count from V_2 that were used in the terminating function (divide) cannot be regained to use in the further aggregation for Q_5, since divide is not invertible.

Computation rules. A view may also be usable to compute the aggregation results in the query without sharing the same aggregate function with the query [16]. For example, the result of an AVG function in the query can be computed by dividing the result of SUM by the result of $COUNT$ from the view. In [17], an aggregate function β is said to be *computed* from a set of aggregate functions $\alpha_1, \alpha_2, \ldots, \alpha_n$ if there is a function g such that for any multiset of values M: $\beta(M) = g(\alpha_1(M), \alpha_2(M), \ldots, \alpha_n(M))$. It can be also written as a *computation rule*: $\alpha_1, \alpha_2, \ldots, \alpha_n \rightarrow \beta$. For instance, there is a computation rule from SUM and $COUNT$ to AVG, i.e. $SUM, COUNT \rightarrow AVG$.

The authors in [16] and [17] consider aggregate function requirements for potentially valid views to rewrite a query by combining the properties mentioned above, which are adapted below for data citation:

Definition 4. **Aggregate function requirements** Suppose a query Q has an aggregate function α, which takes a set of variables X as arguments. If M is *valid* for some query tuples, the aggregate functions in V should satisfy the following conditions under view mapping $M = (h, \phi)$:

1. V also has an aggregate function α with arguments Y, and $\phi(Y) = X$ OR there exists some *computation rule* $\beta_1, \beta_2, \ldots, \beta_m \rightarrow \alpha$ and $\beta_1, \beta_2, \ldots, \beta_m$ also appear in the head of V, all of which take same set of variables Y as arguments and $\phi(Y) = X$.

2. If V has finer granularity than Q, then the functions α or $\beta_1, \beta_2, \ldots, \beta_m$ must also be invertible.

In this case, we say that the aggregate term $\alpha(X)$ in Q is *covered* under view mapping M.

4.3.2 Valid view mappings for aggregate queries

We can now formally provide conditions for valid view mappings for aggregate queries, which are still composed of *schema-level conditions* and a *tuple-level condition.*

Definition 5. **Schema-level conditions for aggregate queries.** Given an aggregate query Q and a view mapping $M = (h, \phi)$ from view V to Q. The *schema-level conditions* are as follows:

1. For *grouping variables* of Q, the following must hold:
   (a) If V is a *conjunctive* view, then for every grouping variable X of Q there is a head variable Y in V such that $\phi(Y) = X$.
   (b) If V is an *aggregate* view, then V must have the same or finer granularity than Q under M.
2. There exists at least one *aggregate term* with aggregate function α taking a set of variables X' as arguments in the head of Q such that:
   (a) If V is a *conjunctive* view, then there is a set of head variables Y' in V such that $\phi(Y') = X'$.
   (b) If V is an *aggregate* view, then Q and V should satisfy the conditions in Definition 4.

Suppose the schema-level conditions are satisfied for a view mapping M. M is a *valid view mapping* for some query tuples iff the following *tuple-level condition* holds:

Definition 6. **Tuple-level condition for aggregate queries.** Let $t \in Q(D)$ with how-provenance polynomial W. Furthermore, given a multiset $\{t_1, t_2, \ldots, t_p\} \in V(D)$, let $t_i (i = 1, 2, \ldots, p)$ have a how-provenance polynomial $W'_i = W'_{i_1} + W'_{i_2} + \cdots + W'_{i_q}$. If for $\{t_1, t_2, \ldots, t_p\}$ and t, the following condition holds, then we say that M is valid for t (not for a single how-provenance monomial): Every monomial W'_{i_j} in $\sum_{i=1}^{p} W'_i$ can be *mapped* to some monomial in W as a one-to-one function under M.

Example 6. Continuing Example 2, recall Q_2, V_2, and view mapping $M_{22} = (h_{22}, \phi_{22})$. In terms of *schema-level conditions*, M_{22} is satisfied for all query tuples because 1) V_2 has the *same granularity* as Q_2 under M_{22}; and 2) the aggregate term of V_2, G, can be mapped to the aggregate variable of Q_2, Gid, which also shares the same aggregate function $COUNT$ and thus satisfies Def. 4.

However, the *tuple-level condition* does not hold for the query tuple $t_{q_2 2}$. If we compare its provenance (i.e. $W = g_2 + g_3$) to the provenance polynomial of the view tuple $t_{v_2 1}$ (i.e. $W' = g_2 + g_3 + g_4$), the monomial mapping between W' and W is not a one-to-one function since g_4 is missing from the mapping. Note that if the predicate in Q_2, $Gid \leq 3$, is relaxed to $Gid \leq 4$ then the token g_4 appears in W and the monomial mapping between W' and W is one-to-one. However, if the predicate is further relaxed to $Gid \leq 5$, then g_5 is included in W and the tuple-level condition is again violated since g_5 is not in W'. This reasoning is significantly more complicated than that in Example 4 since the validity of view mappings is determined by comparing entire how-provenance polynomials between the query tuple and view tuples instead of single how-provenance monomials.

Finally, as in [40], *covering sets* are computed which cover as many *aggregate terms* and *relational subgoals* in the query as possible using the fewest view mappings, i.e. that are *maximal and non-redundant*.

Example 7. Again revisiting Example 2, if we omit the predicates of V_1 and V_2 then M_{12} and M_{22} are valid for both $t_{q_2 1}$ and $t_{q_2 2}$. There are two covering sets, $\{M_{12}\}$ and $\{M_{22}\}$, since both M_{12} and M_{22} cover the aggregate term ($COUNT(Gid)$) and the subgoal $Gene(Gid, Name, T)$ of Q_2. The view mapping combination $\{M_{12}, M_{22}\}$ is redundant since it covers the same terms of Q_2 as its subset $\{M_{12}\}$ and $\{M_{22}\}$.

5. IMPLEMENTATION

Generating provenance-based citations for aggregate queries and views relies on ideas from query rewriting using views as well as provenance. However, bringing those ideas from theory into practice raises several engineering challenges. We now discuss those challenges, starting by analyzing the algorithmic complexity of ProvCite before moving to implementation details and optimizations used in our system. In what follows, we assume that the underlying database system is *provenance-enabled*.

5.1 Algorithmic complexity

An overview of our implementation is shown in Algorithm 1. We discuss the cost of each of these three steps in turn.

Algorithm 1: Overview of PBA

Input : a set of views: $\mathcal{V} = \{V_1, V_2, ..., V_k\}$, user query: Q, a Database instance D
Output: Covering sets for every query tuple in $Q(D)$
1 *Preprocessing step*: Return a set of all possible view mappings $\mathcal{M}$ and the provenance of Q
2 *Reasoning step*: Under a view mapping M from V to Q, determine the validity of M for each query tuple by comparing the provenance between Q and V
3 *Covering sets step*: Calculate covering sets by combining valid view mappings for each query tuple.

Preprocessing step. The major overhead is retrieving the query provenance, which is determined by the underlying provenance-enabled database.

Reasoning step. Let N_{pv} be the total number of how-provenance monomials in the view instance and N_{pq} be the total number of how-provenance monomials in the query instance (this is also the number of tuples before aggregation). Then the time to check the validity of a view mapping is $O(N_{pq} + N_{pv})$ since every how-provenance monomial in the view instance is compared to some how-provenance monomial in the query instance. If there are m view mappings, then the overall time complexity for this step is $O(m * N_{pq}) + O(m * N_{pv})$. Suppose k is an upper bound on the number of relational subgoals in the query or view body, and the largest relation in the database has n tuples. Then the time complexity becomes $O(m * n^k)$. Our experiments with realistic queries in Section 6, however, show that in practice the performance is still acceptable since both N_{pq} and N_{pv} are typically not very large (less than 1 million).

Covering set step. The time for this step depends on the *policies* defined by the DBA on how to convert *covering sets* into formatted citations (see [40]). In the worst case, the number of coverings sets may be exponential in m and all covering sets are used in a rewriting. However, an order of magnitude speed-up can be achieved using the optimization strategies described next (see experimental results in Section 6). In practice, we also believe that a "minimal cost" policy will be used to generate concise citations rather than an expensive "use all" policy, in which case the covering sets are pruned, resulting in cost which is linear in m.

5.2 Optimization and implementation

Our worst case complexity analysis shows that it is challenging to generate fine-grained citations with acceptable performance if implemented naively. We therefore use a

number of optimizations in the last two steps. In the *Reasoning* step, the query provenance is indexed and the view provenance is materialized. In the *Covering sets* step, covering sets are represented as bit arrays and the order in which view mapping sets are combined is optimized using a clustering algorithm called *Affinity propagation* [23]. We discuss the details of these optimizations below, after introducing an example to illustrate how ProvCite is implemented.

Example 8. Given the views $V_1 - V_6$ defined in Section 3.1, suppose the query is as follows:

$Q(G, COUNT(T), MAX(L), COUNT(E)) :-$
$Exon(E, L, T'), Transcript(T, N, Ty, G), T = T', E <= 3$

In the *pre-processing* step, the provenance of the query is retrieved. Using the instances of *Exon*, *Gene* and *Transcript* shown in Tables 1-3, the instance of Q along with the how-provenance polynomials is shown in Table 9.

Table 9: $Q(D)$ with how-provenance polynomials

	G	COUNT(T)	MAX(L)	COUNT(E)	prov
t_{q1}	1	1	1	1	$e_1 * r_1$
t_{q2}	2	2	3	2	$e_2 * r_2 + e_3 * r_2$

Table 10: View mappings from $V_3 - V_6$ to Q

view mapping	aggregate terms covered	relational subgoals covered
M_3 (1000)	COUNT(T),MAX(L), COUNT(E) (111)	Transcript(T, N, Ty, G), Exon(E, L, T') (11)
M_4 (0100)	COUNT(T) (100)	Transcript(T, N, Ty, G), Exon(E, L, T') (11)
M_5 (0010)	MAX(L), COUNT(E) (011)	Transcript(T, N, Ty, G), Exon(E, L, T') (11)
M_6 (0001)	COUNT(T) (100)	Transcript(T, N, Ty, G) (01)

Next, all possible *view mappings* are constructed. We can find four view mappings, $M_3 = (h_3, \phi_3)$, $M_4 = (h_4, \phi_4)$, $M_5 = (h_5, \phi_5)$ and $M_6 = (h_6, \phi_6)$, under which $V_3 - V_6$ are mapped to Q respectively. $M_3 - M_5$ have the same form, where $h_3 - h_5$ is $\{Transcript(T1, N1, Ty1, G1) \rightarrow Transcript(T, N, Ty, G), Exon(E, L, T2) \rightarrow Exon(E, L, T')\}$ and $\phi_3 - \phi_5$ are the induced variable mappings. In contrast, M_6 only maps the subgoal $Transcript(T, N, Ty, G)$ from V_6 to $Transcript(T, N, Ty, G)$ from Q. Note that all the schema-level conditions are independent of the individual query tuples, and can be used to remove invalid view mappings early. In this example, $M_3 - M_6$ all satisfy the *schema-level conditions*, since under each mapping all the grouping variables and at least one aggregate term of Q are covered. Table 10 shows how each view mapping covers the aggregate terms and relational subgoals of Q (ignore the bit arrays for now).

In the *Reasoning* step, since V_3 is a conjunctive view, the validity of M_3 for a query tuple t_q only depends on the *existence* of t_q in $V_3(D)$ under mapping M_3 (Section 3.2). So we can simply retrieve the base relation tuple for each provenance token appearing in the query to evaluate the view predicates. For example, the validity of M_3 can be checked simply by examining the predicates of V_3. Since the first predicate in V_3, $T = T'$, is also in Q, every tuple in the query instance must satisfy it. However, the second predicate, $E \leq 2$, can affect the existence of query tuples in V_3. Table 1 shows that only the tuples with how-provenance

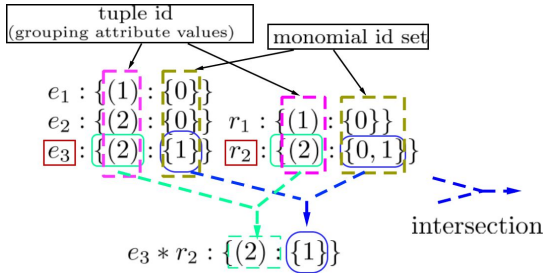


Figure 1: Query provenance index for Q and how to compute coordinate for $e_3 * r_2$ from $V_4(D)$

Table 11: $V_4(D)$ with how-provenance polynomials

	G1	COUNT(T1)	prov
$t_{v_4 1}$	1	1	$e_1 * r_1$
$t_{v_4 2}$	3	1	$e_3 * r_2 + e_4 * r_2$

Table 12: $V_5(D)$ with how-provenance polynomials

	G1	MAX(L)	COUNT(E)	prov
$t_{v_5 1}$	1	1	1	$e_1 * r_1$
$t_{v_5 2}$	2	3	2	$e_2 * r_2 + e_3 * r_2 + e_4 * r_2$

Table 13: $V_6(D)$ with how-provenance polynomials

	G	COUNT(T)	prov
$t_{v_6 1}$	1	1	r_1
$t_{v_6 2}$	2	1	r_2

Table 14: $Q(D)$ with valid view mappings and covering sets (aggregate terms omitted)

	G	valid view mappings	covering sets
t_{q1}	1	M_3, M_4, M_5, M_6	$\{\{M_3\}, \{M_4, M_5\}, \{M_5, M_6\}\}$
t_{q2}	2	M_6	$\{\{M_6\}\}$

tokens e_1 and e_2 satisfy $E \leq 2$. Thus M_3 is only valid for t_{q1}, whose how-provenance polynomial only includes e_1. Note that the implementation used here is different from RBM since evaluating view predicates is achieved by referencing the base relation tuples with the query provenance rather than computing extra predicates in the query evaluation.

In contrast, since V_4, V_5 and V_6 are aggregate views, we need to compare their how-provenance expressions with the how-provenance of the query to check the tuple-level conditions. That can be naively implemented by scanning the entire query provenance for *every view mapping* to build satisfiable provenance mappings (Def. 6), which is expensive when N_{pv}, N_{pq} and the number of view mappings m are large. To reduce this cost, we index the query provenance.

Query provenance index optimization. In order to avoid multiple scans over query provenance, we build an index I for each token in the query provenance to indicate which query tuples (represented by grouping variable values) and which provenance monomials the token is in. The provenance index for Q is shown in Figure 1. For example, referencing Table 9, note that token r_2 is in the 0^{th} and 1^{st} monomial in the query tuple t_{q2}, which has value 2 for the grouping variable G. So the index for r_2 is $r_2 : \{(2) : \{0, 1\}\}$, where (2) represents the tuple id while $\{0, 1\}$ is the monomial id set. For a how-provenance monomial in the view, e.g. $e_3 * r_2$ in $t_{v_4 2}$ (with grouping variable value 2), to determine

whether it can be mapped to some query provenance monomial, we can retrieve the index for e_3 and r_2 with grouping variable value 2 respectively, i.e. $\{1\}$ and $\{0,1\}$, and take the intersection, i.e. $\{1\}$. This indicates that e_3 and r_2 coexist in the 1^{st} monomial of the query tuple with grouping variable value 2 (i.e. t_{q2}). This derivation process is highlighted in Figure 1.

We can further optimize the intersection operation by representing the monomial ids with bit arrays, where the i^{th} bit is 0/1 iff a token is/isn't in the i^{th} monomial, and applying bit AND operations. This strategy only requires one full scan over the query provenance to build an index for all view mappings. Details on how to use the index to determine whether provenance mappings from view tuples to query tuple satisfy Def. 6 are presented in Algorithm 2.

Algorithm 2: Checking provenance mappings

```
Input  : a view V, a query Q, a view mapping M from
         V to Q, query tuple t_q ∈ Q(D), query
         provenance index I, a set of view tuples
         T_v ⊆ V(D)
Output: Whether provenance mappings from T_v to t_q
         satisfy Def. 6
for t_v ∈ T_v do
    Retrieve the provenance monomial set P̄ of t_v
     Retrieve the grouping variable values (Gv) of Q
     under mapping M
    for each provenance monomial P ∈ P̄ do
        for each provenance token p ∈ P do
            if p is not in I OR Gv is not in the entry of
             I for p then
                return false
            end
        end
        Perform intersection over the index for p with
         grouping variable values Gv
        if the intersection result is empty then
            return false
        end
    end
end
return true
```

Materialization and parallelism optimization. To further improve performance, the provenance of the aggregate views along with the view content can be materialized before the query arrives. The strategy with materialized view provenance is called *eager*, whereas that without is called *lazy*. The *eager* and *lazy* strategies are compared in Section 6. We observe that reasoning about the validity of view mappings is highly parallelizable since the reasoning between different view mappings is independent. However, since fully parallel computation in a single machine can incur large memory consumption, ProvCite only processes five view mappings at a time. Exploring how to fully develop our system in a distributed environment is left for future work.

Using the instances and provenance expressions of $V_4 - V_6$ presented in Tables 11-13, the valid view mappings for every query tuple are presented in Table 14. Note that for tuple t_{q2}, although all of its how-provenance monomials exist in the view tuple $t_{v_5 2}$, it does not include $e_4 * r_2$ which is used to construct $t_{v_5 2}$, violating the tuple-level condition. Intuitively, since the value of the aggregate term may come from this component of the monomial $(e_4 * r_2)$, $t_{v_5 2}$ should not provide citation information for t_{q2}.

Algorithm 3: Compute covering sets

```
Input  : a set of valid view mappings M for query
         tuple t ∈ Q(D), query Q
Output: a set of covering sets C
For each aggregate term of Q, derive a set of view
 mappings covering it, which forms an array of view
 mapping sets S.
Determine the order to compute the cross product of
 every element in S
Initialize C as the first view mapping set s_0 from S.
for each set s ∈ S − {s_0} do
    Initialize cross product result C' = {}:
    for each view mapping set M̄' ∈ C do
        for each view mapping M ∈ s do
            get three bit arrays of M̄' (M): b_1 (b'_1),
             b_2(b'_2) and b_3 (b'_3)
            construct new view mapping set M̄'' based
             on the bit OR operation result
             b_i ∨ b'_i(i = 1, 2, 3) and put it into C'
        end
    end
    Remove duplicates from C'
    C = C'
end
```

Finally, valid view mappings are shown in Table 14 along with the query instance, which are then used to compute covering sets for each query tuple in the *Covering sets step*.

As we observed in [40], it is time-consuming to compute *all* covering sets. We therefore use two new strategies to achieve speed-up: 1) representing coverings sets using bit arrays; and 2) applying clustering algorithms to avoid an explosion of intermediate results. These lead to an order of magnitude performance gain (see Section 6).

Bit array optimization. The computation of covering sets involves merging valid view mappings and removing duplicates, which can be optimized using bit operations. For example, for Q, the aggregate term $COUNT(T)$, $MAX(L)$ and $COUNT(E)$ are covered by $\{M_3, M_4, M_6\}$ (denoted by S_1), $\{M_3, M_5\}$ (denoted by S_2) and $\{M_3, M_5\}$ (denoted by S_3) respectively (see Table 10). We can encode the $0^{th} - 3^{rd}$ view mappings $M_3 - M_6$ as $\{0,1,2,3\}$, the $0^{th} - 2^{nd}$ aggregate terms ($COUNT(T)$, $MAX(L)$ and $COUNT(E)$) as $\{0,1,2\}$, and the $0^{th} - 1^{st}$ relational subgoals ($Exon$ and $Transcript$) as $\{0,1\}$. In this manner, arbitrary view mapping combinations (and thus covering sets) can be represented using three bit arrays in which the i^{th} bit is 1 (0) iff the i^{th} view mapping is included (missing), or the i^{th} aggregated term or relational subgoal is covered (not covered). For example, M_4 is the 1^{st} view mapping, represented by 0100 (the leftmost bit is the 0^{th} bit). M_4 covers the 0^{th} aggregate term ($COUNT(T)$) and the 0^{th} and 1^{st} relational subgoals, which are represented by bit arrays 100 and 11 respectively. The bit array representations for other view mappings are listed in Table 10. To compute covering sets, the view mapping combinations from the cross product of $\{S_1, S_2, S_3\}$ (denoted by $S_1 \times S_2 \times S_3$) are considered, which are constructed by applying bit OR operations over the bit arrays from those view mappings. For example, referencing Table 10, the covering set $\{M_4, M_5\}$ can be constructed by unioning bit arrays 0100 and 0010, and the aggregate terms (relational subgoals resp.) jointly covered by them are computed by unioning 100 and 011 (11 and 11 resp.).

The pseudocode for computing covering sets using bit array representations is presented in Algorithm 3.

Clustering algorithm optimization. Since cross product ($\times$) is commutative and associative, different orderings of operands result in the same output but may incur different overhead. For example, with $S_1 \times S_2 \times S_3$, if $S_2 \times S_3$ is computed first, the result is $\{M_3, M_5\}$, $\{M_5, M_5\}$, $\{M_3, M_3\}$, $\{M_5, M_3\}$. After removing obvious redundancy, the result is $\{M_3, M_5\}$, $\{M_5\}$, $\{M_3\}$. Note that $\{M_3, M_5\}$ is a duplicate compared to $\{M_3\}$ since 1) $\{M_3, M_5\}$ and $\{M_3\}$ cover the same set of aggregate terms and relational subgoals (checked by comparing the corresponding bit arrays); and 2) $\{M_3\}$ is a subset of $\{M_3, M_5\}$. It is therefore safe to remove $\{M_3, M_5\}$ since in the final result, any view mapping combinations which include $\{M_3, M_5\}$ will be a duplicate compared to one that includes $\{M_3\}$ and thus won't be a covering set. So the intermediate result of $S_2 \times S_3$ is $\{\{M_3\}, \{M_5\}\}$, which is smaller than the result of the other pairs. This is due to the high similarity between S_2 and S_3 (actually $S_2 = S_3$). To find good orderings for computing the cross product such that the intermediate result is minimized, clustering algorithms are applied so that view mapping sets which are similar to each other can be clustered and merged first (e.g. S_2 and S_3). In ProCite, the affinity propagation clustering algorithm [23] is used since it does not require a pre-specified number of clusters.

6. EXPERIMENTS

We start by describing the datasets and workloads used before presenting the experimental results.

6.1 Experimental set-up

ProvCite was implemented in Java 8 using PostgreSQL 9.6.3 as the underlying DBMS. All experiments were conducted on a Linux server with an Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and 64GB of main memory. Although there are provenance tools which support aggregate queries for relational database systems, e.g. GProM [9], they are overly complex for our purposes, and become a bottleneck for interactive computation. We therefore implemented a provenance layer from scratch, which simply collects how-provenance [27, 8] for each query tuple.

Datasets. Two realistic datasets were used in addition to GENCODE: Hetionet[1] and DBLP-NSF[2][4]. Summary information about the three datasets, including the number of relations, average size per relation, and the size of the largest relation is presented in Table 15.

We converted Hetionet, which is stored in Neo4j, into a relational database[3]. DBLP-NSF integrates DBLP publication information with NSF award information to augment traditional paper citations with funding information, and was developed in [40].

Workloads. We test the performance of ProvCite using both *synthetic* and *realistic* workloads. As mentioned earlier, to retrieve the provenance of aggregate views we can use either the *eager* or *lazy* strategy; we can also build an *index* on query provenance. We measure the performance gains using the eager strategy and index separately under both workloads. The performance also depends on the *policies* used; different policies can lead to different results, and can generate either all or some of the covering sets. Due to space, only the case where *all* the covering sets are generated is presented here.

The purpose of using *synthetic workloads* is to determine the key factors which influence performance. Extensive experiments were performed in [40] measuring the *total reasoning time* to generate the covering sets (t_{cs}) and the *citation generation time* after covering sets are constructed. The *citation generation time* is not considered here since ProvCite only changes how *valid view mappings* are determined during covering sets construction process relative to the implementations of RBM. Since ProvCite relies on the query provenance for reasoning, the query time over the provenance-enabled database (t_q) is also recorded. The total execution time t_{total} is measured as well, which is $t_q + t_{cs}$.

In [40], t_{cs} primarily depends on: 1) the number of view mappings (denoted N_v); 2) the total number of predicates under the view mappings (denoted N_p); and 3) the size of the query instance before duplicates are removed (which is the same as the total number of how-provenance monomials in the query instance N_{pq}). The experiments measure the effect of these metrics on t_{cs}. The total number of how-provenance monomials in the view instance on average (N_{pv}) can influence performance according to the analysis in Section 5, and its effect is also considered in the experiments. So for the experiments for *synthetic workloads*, a query generator is built, which can generate random aggregate queries over GENECODE with N_{pq} how-provenance monomials in total. Given a query Q from this query generator, a view generator can generate N_v views, each of which has N_{pv} provenance monomials in total and has exact one view mapping to Q.

The trade-offs between ProvCite and two implementations of RBM proposed in [40], TLA and SSLA, are also measured. As mentioned before, RBM can be extended to handle aggregate queries when views are conjunctive views (but not aggregate views). In this case, TLA, SSLA and ProvCite all generate the same final, fine-grained citations.

In the *realistic workloads*, we use frequent queries against the three databases, and build views to represent the portions of data in the database associated with predefined citations. Both the synthetic and realistic views and queries used are available in our Github repository[4].

To represent the summary information provided by GENCODE, we defined aggregate views to compute the total number of transcripts per gene, and the total number of exons per gene and per transcript. Two additional parameterized views are also defined to represent basic information (e.g. ID, name and type) for each transcript and gene, respectively. The realistic queries compute the total number of exons ($q1$) and the total number of transcripts per type of gene ($q2$) respectively.

For DBLP-NSF we use the realistic views defined in [40]. We also add aggregate views to reflect publicly available statistics related to this database, such as the total number of publications per faculty member[5] and the total number of grants per institution[6]. Some realistic aggregate queries

[1] `https://neo4j.het.io/browser/`

[2] `https://data.mendeley.com/datasets/ycnngyv5bd`

[3] Available at `https://github.com/thuwuyinjun/Data_citation_provenance/files/2417454/hetionet_postgresql.zip`

[4] `https://github.com/thuwuyinjun/Data_citation_provenance`

[5] `http://csrankings.org/`

[6] `https://dellweb.bfa.nsf.gov/awdlst2/default.asp`

are designed to represent other summary information, such as the total number of publications per institution ($q3$) and total amount of grants per state ($q4$).

Hetionet integrates information from various resources, and includes information about genes, biological process, drugs, etc. This information is stored in different relations in the database. Of these, the biological process relation is associated with citation information (i.e. related publication IDs). After consulting with the authors of Hetionet, two views were defined. The first one is a parameterized view showing the biological processes that a particular gene is involved in. The second counts the total number of connections between each biological process and corresponding genes by joining several relations, such as the biological process and gene relations. A typical query ($q5$) counts the total number of connections between each biological process and a certain drug via some genes.

Table 16 provides a summary of the notation mentioned in this subsection.

6.2 Experimental results

We now report on results from the synthetic and realistic workloads.

6.2.1 Synthetic workloads

We measured the impact of the provenance size of the query and view instances on time performance (Exp1), and the relative performance of ProvCite and two implementations of RBM, TLA and SSLA, while varying the view mapping number (Exp2) and the view predicate number (Exp3).

Exp1. This experiment measures how the total execution time (t_{total}) is influenced by the total number of how-provenance monomials in the query instance (N_{pq}) as well as in the view instance (N_{pv}). We randomly generate an aggregate query, and vary N_{pq} by adding appropriate predicates. A fixed number of aggregate views are also generated such that there is exact one view mapping from each of them to the query and the total number of view mappings is fixed at 20. In practice, the number of views that touch the query is usually far smaller than the total number of views, so 20 is a pretty large number. Both N_{pq} and N_{pv} are varied from 50K to 5M. The total time is measured for different (N_{pq}, N_{pv}) pairs under the *eager* and *lazy* strategy with the query provenance index, and the *lazy* strategy without the index.

Results. The results are shown using 3D surfaces in Figure 2a, with the eager strategy with index, lazy strategy with index and pure lazy strategy shown in red, green and yellow respectively. The query time t_q is also recorded in black. It shows that the query provenance index leads to about 1.0x-1.8x speed-ups in most cases by comparing lazy strategy with index and pure lazy strategy, while materializing view provenance results in about 1.1x-1.5x speed-ups by comparing eager strategy and lazy strategy with index. The combination of the index and eager strategy leads to up to 2x performance gains. The result also shows the *scalability* of our approach since it takes less than 2 mins to process a query instance with up to 5 million how-provenance monomials and 20 views with up to 5 million how-provenance monomials, which rarely happens in practice.

Exp2. The goal of this experiment is to compare the relative performance of ProvCite, TLA and SSLA while varying the number of view mappings (N_v). Since TLA and SSLA cannot handle aggregate views, only conjunctive views are used. In this case, the provenance of views is not necessary; there is no difference between the eager and lazy strategy and the query provenance index is not useful. However, the two optimization strategies on covering set computation, i.e. applying bit arrays and clustering algorithms, are useful and are measured here. The query is a fixed aggregate query with 1 million how-provenance monomials in its instance. N_v is varied from 1 to 50 and there are no predicates or lambda variables for each individual view.

Results. The experimental results are presented in Figure 2b, which shows the change of t_{total} for ProvCide and the number of covering sets (N_{cs}) as the number of view mappings (N_v) increases, with and without using bit arrays and clustering algorithm. TLA and SSLA have almost the same performance as ProvCite, and are not shown. Figure 2b shows that when N_v is large, an exponential number of covering sets are generated, leading to bad performance (see blue line). Bit array computations and the use of clustering leads to about a 5x and 2x speed-up respectively; an order of magnitude performance gain is achieved by combining both.

Exp3. In this experiment, ProvCite is compared with TLA and SSLA while varying the total number of predicates (N_p) in views. Similar to Exp2, the query is an aggregate query which can generate about 1 million tuples. The number of view mappings is fixed at 10 and there are initially no predicates. In each run, one more local predicate is added. As shown in [40], increasing N_p significantly influences the query time and hence performance of TLA and SSLA since the query is extended to evaluate the view predicates.

Results. The results are shown in Figure 2c. As the number of predicates increases, $t_{total} = t_{cs} + t_q$ increases

Table 15: Summary of datasets

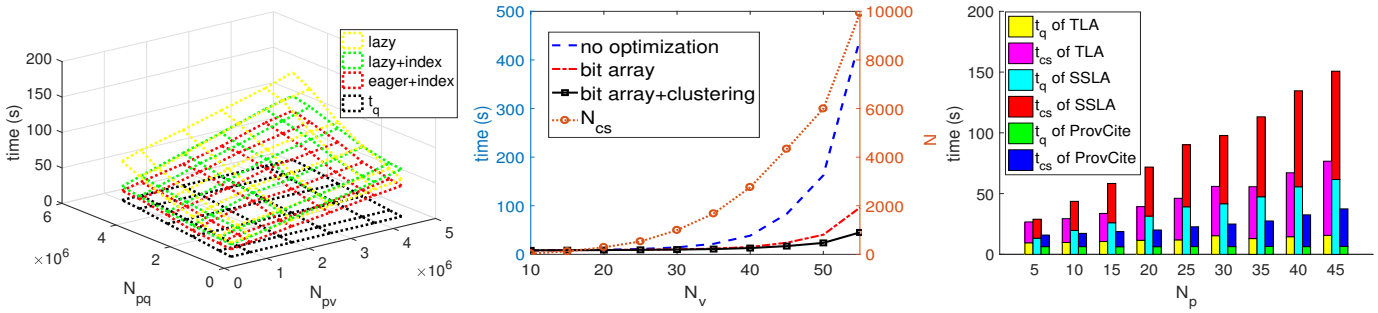
Dataset name	relation #	average tuple # per relation	tuple # of largest relation
GENECODE	7	600k	2000k
Hetionet	38	60k	500k
DBLP-NSF	17	600k	6000k

Table 16: Notation used in the experiments

Notation	Meaning
t_{cs}	total reasoning time to generate the covering sets for all query tuples
t_q	query time over the database
t_{total}	total execution time
N_v	total number of view mappings
N_p	total number of predicates under the view mappings
N_{pq}	total number of how-provenance monomials in the query instance
N_{pv}	total number of how-provenance monomials in the view instance on average

Table 17: Experimental results on realistic datasets

Query	t_{total} (s) (eager + index)	t_{total} (s) (lazy + index)	t_{total} (s) (lazy)	N_{pq}	N_v	N_p	t_q(s)
$q1$	11.05	12.93	11.89	1237k	1	0	5.09
$q2$	1.75	2.06	3.26	203k	2	0	0.69
$q3$	4.95	6.62	6.44	507k	2	0	2.92
$q4$	5.90	6.49	6.33	416k	1	0	2.80
$q5$	4.65	5.10	4.81	243k	3	0	2.32



(a) t_{total} **with varied** N_{pq} **and** N_{pv} **(b)** t_{total} **with varied** N_v **(c)** t_{total} **with varied** N_p

Figure 2: Experimental results for synthetic workloads

slowly for ProvCite. In contrast, TLA and SSLA are twice as slow as ProvCite for large N_p. To understand the reason for this, the query time for TLA, SSLA and ProvCite is also presented in this figure, showing that the increasing query time becomes the major overhead for both TLA and SSLA.

Discussion. The experiments reveal that all four metrics, N_{pq} (the total number of how-provenance monomials in the query instance), N_{pv} (the total number of how-provenance monomials in the view instance on average), N_v (the total number of view mappings), and N_p (the total number of predicates under the view mappings) can affect the total reasoning time t_{cs}. In extreme cases, where the value of the metric is very large, the performance of ProvCite is bad especially if implemented naively. However, the performance can be substantially improved using the bit array and clustering optimization strategies. We therefore expect ProvCite to have acceptable time performance for realistic workloads, where the extreme cases rarely arise.

6.2.2 *Realistic workloads*

The experimental results for realistic workloads are presented in Table 17, which includes the total execution time (t_{total}) for three cases (lazy, lazy + index and eager + index), as well as the metrics that can potentially affect the performance: the total number of how-provenance monomials in the query instance (N_{pq}), the total number of view mappings (N_v), the total number of predicates in the views under all the view mappings (N_p) and the time to query the provenance along with the instance. Except for $q1$, t_{total} is less than 10 seconds for all queries. Although N_{pq} is more than one million in $q1$, t_{total} is only about 11-13 seconds for the three strategies, which is acceptable considering the large query instance. Note that the index does not always help since it may take significant time to build the index for query provenance (e.g. up to 3 seconds for $q1$) and its performance gain is not significant in the case of small number of view mappings.

However, as shown in Section 6.2.1, the index provides scalability especially in the extreme cases. We also list the query time over the provenance-enabled database in the last column, which indicates that the reasoning time ($t_{cs} = t_{total} - t_q$) is almost the same as t_q. Thus, while users are browsing the query result, the system can generate covering sets for all query tuples in the background, and instantly construct formatted citations upon tuple selection.

Discussion. The experimental results show that reasonable time performance can be guaranteed in practice where none of the crucial metrics become too large. Revisiting the experimental results for synthetic workloads, when the total number of how-provenance monomials in the query instance (N_{pq}) is about 1 million (as in $q1$), t_{total} is about 1 min in the case of large N_{pv} and N_v values. However, since only one view mapping appears for $q1$, the time shown for $q1$ is significantly smaller. It also indicates that the number of view mappings should be small since views associate *different parts* of the database with citations and thus only a small portion of them touch the query result. Performance is therefore acceptable in practise.

7. CONCLUSIONS

We build on the connection to *data provenance* to develop a model for data citation which is able to handle aggregate queries and views. The model reasons about citations at the level of tuples in the query result using provenance to enable citations to arbitrary subsets of the query result (fine-grained citation). The Provenance-Based Model was implemented in ProvCite, which runs on top of a provenance-enabled RDBMS. Extensive experiments were conducted under both *synthetic* and *realistic workloads*, and show that ProvCite can not only handle a larger class of queries than the Rewriting-Based Model [40] (which assumes conjunctive queries and views), but is much faster in some cases.

Optimizations used in ProvCite include materializing view provenance, building an index to speed up comparisons between the query and view provenance, using bit arrays, and using clustering algorithm to determine the best order of combining view mappings. Several of these are applicable in other scenarios where the provenance from different queries or views needs to be compared, such as fine-grained access control [26] and linked brushing in data visualization [37].

In future work, we would like to explore how to insert data citation into the larger citation ecosystem involving bibliometrics. We would also like to explore how to combine data citation with black-box computations, including Machine Learning models.

8. ACKNOWLEDGMENTS

This research was partially funded by NSF IIS 1302212 NSF ACI 154736, then Israel Innovation Authority, the Israel Science Foundation, and Len Blavatnik and the Blavatnik Family foundation. The authors thank Peter Buneman for conceiving the idea of data citation, and Val Tannen for discussions on query rewriting and provenance.

9. REFERENCES

[1] *Out of Cite, Out of Mind: The Current State of Practice, Policy, and Technology for the Citation of Data*, volume 12. CODATA-ICSTI Task Group on Data Citation Standards and Practices, 2013.

[2] DataCite Metadata Schema Documentation for the Publication and Citation of Research Data, v4.0. Technical Report, DataCite Metadata Working Group, 2016.

[3] F. N. Afrati, C. Li, and J. D. Ullman. Using views to generate efficient evaluation plans for queries. *Journal of Computer and System Sciences*, 73(5):703–724, 2007.

[4] A. Alawini, S. Davidson, S. Pandey, G. Silvello, and Y. Wu. "DBLP-NSF dataset SQL dump", Mendeley Data, v5. `http://dx.doi.org/10.17632/ycnngyv5bd.5`.

[5] A. Alawini, S. Davidson, G. Silvello, V. Tannen, and Y. Wu. Data citation: A new provenance challenge. *Data Engineering*, page 27, 2018.

[6] A. Alawini, S. B. Davidson, W. Hu, and Y. Wu. Automating data citation in CiteDB. *PVLDB*, 10(12):1881–1884, 2017.

[7] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. *ACM Transactions on Database Systems (TODS)*, 37(4):30, 2012.

[8] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 153–164. ACM, 2011.

[9] B. S. Arab, S. Feng, B. Glavic, S. Lee, X. Niu, and Q. Zeng. GProM-a swiss army knife for your provenance needs. *Data Eng. Bull*, 41(1):51–62, 2018.

[10] J. Brase, I. Sens, and M. Lautenschlager. The Tenth Anniversary of Assigning DOI Names to Scientific Data and a Five Year History of DataCite. *D-Lib Magazine*, 21(1/2), 2015.

[11] P. Buneman, S. B. Davidson, and J. Frew. Why data citation is a computational problem. *Communications of the ACM (CACM)*, 59(9):50–57, 2016.

[12] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.

[13] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[14] A. Chapman and H. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 523–534. ACM, 2009.

[15] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 190–200. IEEE, 1995.

[16] S. Cohen. User-defined aggregate functions: bridging theory and practice. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM, 2006.

[17] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Transactions on Database Systems (TODS)*, 31(2):672–715, 2006.

[18] S. Cohen, W. Nutt, and Y. Sagiv. Deciding equivalences among conjunctive aggregate queries. *Journal of the ACM (JACM)*, 54(2):5, 2007.

[19] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 155–166. ACM, 1999.

[20] M. P. Consens and A. O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *International Conference on Database Theory*, pages 379–394. Springer, 1990.

[21] S. B. Davidson, P. Buneman, D. Deutch, T. Milo, and G. Silvello. Data citation: A computational challenge. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1–4, 2017.

[22] S. B. Davidson, D. Deutch, T. Milo, and G. Silvello. A model for fine-grained data citation. In *CIDR*, 2017.

[23] D. Dueck and B. J. Frey. Non-metric affinity propagation for unsupervised image categorization. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

[24] FORCE-11. *Data Citation Synthesis Group: Joint Declaration of Data Citation Principles*. FORCE11, San Diego, CA, USA, 2014.

[25] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *ACM SIGMOD Record*, volume 30, pages 571–581. ACM, 2001.

[26] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. Acm, 2006.

[27] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40. ACM, 2007.

[28] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[29] J. Harrow, A. Frankish, J. M. Gonzalez, E. Tapanari, M. Diekhans, F. Kokocinski, B. L. Aken, D. Barrell, A. Zadissa, S. Searle, et al. Gencode: the reference human genome annotation for the encode project. *Genome research*, 22(9):1760–1774, 2012.

[30] D. S. Himmelstein and S. E. Baranzini. Heterogeneous network edge prediction: a data integration approach to prioritize disease-associated genes. *PLoS computational biology*, 11(7):e1004259, 2015.

[31] D. S. Himmelstein, A. Lizee, C. Hessler, L. Brueggeman, S. L. Chen, D. Hadley, A. Green, P. Khankhanian, and S. E. Baranzini. Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *Elife*, 6, 2017.

[32] L. B. Honor, C. Haselgrove, J. A. Frazier, and D. N. Kennedy. Data Citation in Neuroimaging: Proposed Best Practices for Data Identification and Attribution. *Frontiers in Neuroinformatics*, 10(34):1–12, August 2016.

[33] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The ORCHESTRA collaborative data sharing system. *ACM Sigmod Record*, 37(3):26–32, 2008.

[34] J. Klump, R. Huber, and M. Diepenbroek. DOI for Geoscience Data – How Early Practices Shape Present Perceptions. *Earth Science Inform.*, pages 1–14, 2015.

[35] J. McEntyre, U. Sarkans, and A. Brazma. The BioStudies database. *Molecular systems biology*, 11(12):847, 2015.

[36] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, pages 484–495, 2000.

[37] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *PVLDB*, 11(6):719–732, 2018.

[38] N. Simons. Implementing DOIs for Research Data. *D-Lib Magazine*, 18(5/6), 2012.

[39] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, volume 96, pages 318–329, 1996.

[40] Y. Wu, A. Alawini, S. B. Davidson, and G. Silvello. Data citation: Giving credit where credit is due. In *Proceedings of the 2018 International Conference on Management of Data*, pages 99–114. ACM, 2018.

[41] J. Xu, W. Zhang, A. Alawini, and V. Tannen. Provenance analysis for missing answers and integrity repairs. *Data Engineering*, page 39, 2018.

[42] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *ACM SiGMOD Record*, volume 29, pages 105–116. ACM, 2000.