ProvCite: Provenance-based Data Citation

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

Abdu Alawini
UIUC
alawini@illinois.edu

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

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

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

ABSTRACT

A computational challenge associated with data citation is how to automatically generate citations to arbitrary queries against a structured dataset. Previous work has 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. showing 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.

PVLDB Reference Format:
Yinjun Wu, Abdu Alawini, Daniel Deutch, Tova Milo and Susan B. Davidson. ProvCite: Provenance-based Data Citation. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

The amount of information available online in structured datasets is rapidly increasing, 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 scholarly 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 [10]. 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 [28].

In prior work, we proposed a general framework to *automatically generate fine-grained citations for general queries* [4, 35]. The approach is based on a model of *citation views* [10, 21, 20]: Frequent queries are defined as views with associated citations. A query against the database is rewritten in terms of these views, and the associated citations 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* (see Figure 1 for an overview). 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 [35] 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 aggregate (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 possibly involve aggregation. So the techniques of [35] cannot be used.

One such example is Hetionet, a database that "encodes" biology by integrating various types of biological information from different publicly available resources [27]. 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 [26], 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 genes.

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 [37, 34, 23, 16, 15] 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 [35] to use ideas from query rewriting for aggregate queries is possible when views are conjunctive views but still 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 [10, 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. We illustrate how provenance helps in the example below.

Example. (See Figure 1) 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 pulls snippets of information from the database and is formatted by the citation function as {`Group: ‘Jones Group’, Source: ‘HAVANA Project’, ...`}.

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. This is true for the first result tuple in Figure 1, where we assume that gene type TEC only includes genes whose ids are smaller than 50 and has at least 10 such genes. Other tuples in the query result may not appear in V_{gene}, i.e. gene types which include some genes with ids 50 or greater (which we assume for the second result tuple rRNA) or which include fewer than 10 genes. In this case, the tuple would not carry the citation associated with V_{gene}.

Traditional query rewriting using views techniques would conclude that V_{gene} is *not useful* for Q. Furthermore, the RBM tuple-level techniques proposed in [35] could not detect whether V_{gene} is useful for a given tuple in the result

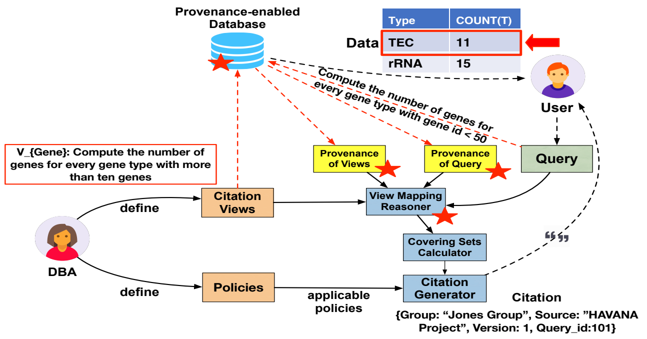

Provenance-enabled Database
Data
Type
COUNT(T)
TEC
11
rRNA
15
User
Compute the number of genes for every gene type with gene id < 50
V_{Gene}: Compute the number of genes for every gene type with more than ten genes
Provenance of Views
Provenance of Query
Query
define
Citation Views
View Mapping Reasoner
DBA
Covering Sets Calculator
define
Policies
applicable policies
Citation Generator
Citation
{Group: “Jones Group”, Source: ”HAVANA Project”, Version: 1, Query_id:101}

Figure 1: System overview of ProvCite

of Q. However reasoning over the *provenance* of result and view tuples could detect that TEC exists in the view instance (by comparing provenance polynomials) and that all tuples in the group have ids smaller than 50 (by finding the gene ids associated with the provenance tokens). Thus, if the user in Figure 1 selects TEC as the subset of interest in the query result, the citation for V_{gene} ({`Group: "Jones Group", Source: "HAVANA Project", ...`}) would be returned.

Approach. Using provenance, we develop a citation system called ProvCite, whose architecture is shown in Figure 1; the key differences between ProvCite and the RBM implementation in [35] are indicated by red stars. The system executes over a provenance-enabled relational database system; here we are using GProM [8]. As in [35], 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 (as illustrated above), 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. So 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. To be practical, the citation should be generated without significantly extending the query time. However, since in the worst case the number of possible covering sets may be exponential in the mapping between the views and the query, the number and size of the views can be large, the number of result tuples can be large, and provenance expressions are big, this would seem to be an impossible task. *Surprisingly, the results of this paper not only show that PBM is feasible and extends results in [35] 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 the PBM called ProvCite, which automatically generates fine-grained citations for the results of general queries, where both the queries and views may involve aggregates. Two strategies are tested for the provenance of views: In the first, provenance is generated on the fly (*lazy strategy*), whereas in the second provenance is pre-computed (*eager strategy*).

4. Experiments using both *synthetic* and *realistic* workloads, comparing ProvCite against RBM approaches [35] in the case of aggregate queries and conjunctive views, and comparing the lazy versus eager strategies for ProvCite when queries and views involve aggregates. The results show that ProvCite has acceptable time performance even when the queries and views have large instances, and can in some cases significantly outperform RBM approaches.

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 libraries community[1, 22] and include: 1) identification and access to the cited data; 2) persistence of the cited data; and 3) completeness of the reference [30, 33, 9, 2]. The community also recognized the importance of citations to aggregate data [1], as have various scientific communities [26, 27, 31]. More recently, data citation has captured the attention of database researchers, who formulated computational challenges [10, 20]. To address these challenges, a model of *citation views* was defined in [21] and implemented in [4, 35]. However, this work was limited to conjunctive queries and views, and did not address 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 [25], and has been extensively studied in the context of conjunctive queries [12, 14, 32, 3] as well as aggregate queries [17, 18]. Various algorithms have been designed to rewrite aggregate queries. For example, [34, 23] provide algorithms for determining whether a materialized view is usable for answering an aggregate query by considering both conjunctive and aggregate views. In [37], 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 [37, 34, 23]; general aggregate functions (such as user defined aggregate functions) cannot be used. The problem of general aggregate functions is considered in [16], and [15] 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.*

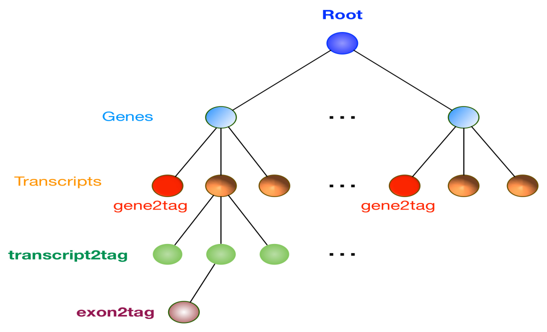


Figure 2: Hierarchical structure in GENCODE

Data Provenance. Data provenance identifies where a piece of data came from and the process by which it arrived in the database [11]. 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* [11], *why-not-provenance* [13], and the *provenance semirings* framework for conjunctive queries [24], aggregate queries [7] and queries with negation [36]. This framework has been used to implement several practical provenance-enabled database systems, such as ORCHESTRA [29] and GProM [8]. The connection between data citation and provenance was discussed in [10] 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* [21], *view mapping* and *validity* of view mappings [35], and then show why the RBM of [35] cannot be extended for aggregate queries and views, motivating the need for *provenance.*

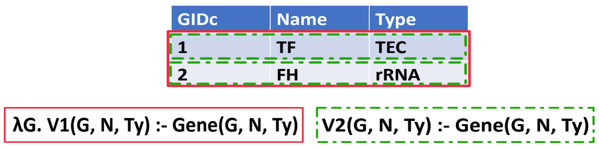
3.1 Running example: GENCODE

We use a simplified schema from GENCODE as our running example. In this database, information is structured hierarchically (see Figure 2): 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 based on this structure is shown below:

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

Relations Gene2tag, transcript2tag and exon2tag capture the annotations (*annot*). The source of the annotation (e.g. a research group or workflow/program) is also stored in GENCODE and can be used for citations. For simplicity, we omit these relations.

Citation views [21] define views of the database to which citations have been specified. A citation view consists of 1)



GIDc
Name
Type
1
TF
TEC
2
FH
rRNA
λG. V1(G, N, Ty) :- Gene(G, N, Ty)
V2(G, N, Ty) :- Gene(G, N, Ty)

Figure 3: Effect of parameters on views

a *view query*, defining the subset of the data to which the citation is attached; 2) a *citation query*, which retrieves information required for the citation for the view; and 3) 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. View queries can be considered to be the *frequent queries* over the database; all other queries will be called *general* queries.

View queries for GENCODE correspond to web-page views created by the DBAs. Each of these views could have an associated citation. For example, the citation query for the web page view of a gene could be used to retrieve the research groups or programs that contributed annotations for the gene; this information, together with the gene name, version of the database, and query used to retrieve the data (e.g. the http address of the web page) could then be formatted by the citation function. Below we show several of these views, which are expressed using S-Datalog [19], an extended version of Datalog that allows aggregates:

$$\begin{array}{ll}
\lambda G.V_1(G,N,Ty) & :-Gene(G,N,Ty)\\
\quad V_2(G,N,Ty) & :-Gene(G,N,Ty)\\
\quad V_3(T1,N1,E,L) & :-Transcript(T1,N1,Ty1,G1),\\
& \quad Exon(E,L,T2), T1=T2,\\
& \quad E>=4\\
\quad V_4(T1,N1,COUNT(E)) & \\
& :-Transcript(T1,N1,Ty1,G1),\\
& \quad Exon(E,L,T2), T1=T2,\\
& \quad L<=2\\
\quad V_5(T1,N1,MAX(L)) & \\
& :-Transcript(T1,N1,Ty1,G1),\\
& \quad Exon(E,L,T2), T1=T2
\end{array}$$

The first three views 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_2 and V_3 are not parameterized, which indicates that the same citation is shared across all the view tuples. Figure 3 shows the effect of the lambda term in V_1 versus the unparameterized view V_2 on a sample instance of Gene. The last two 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_4 could be translated into SQL as:

```
SELECT T.TID, T.Name, COUNT(E.EID)
FROM Transcript T, Exon E
WHERE T.TID=E.TID and E.Level <= 2
GROUP BY T.TID, T.Name
```

V_4 counts the number of exons with level not greater than 2 for each transcript, and includes the transcript name in the result. V_5 returns the maximal level among all exons for each transcript.

3.2 Query rewriting: View mappings

Given a (general) query Q and a set of views $\mathcal{V}$, the RBM implementation in [35] 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*, which are converted to formatted citations later. Each covering set is a *maximal*, *non-redundant* set of *valid* view mappings, i.e. covering 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.

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 there exists a view tuple t' in V such that t' is *visible* in t under M, the reasoning of which depends on examining the lambda variables, the head variables and predicates of views by RBM in [35]. We say that a head variable or a relational subgoal of Q is *covered* by M iff it is involved in M.

Example 1. Suppose a conjunctive view and a conjunctive query are defined below:

$$\begin{array}{l}
V_1'(E,T):-Exon(E,L,T), E<=4\\
Q_1'(Eid,Tid):-Exon(Eid,Level,Tid)
\end{array}$$

There is a view mapping $M_1'=(h_1',\phi_1')$ from V_1' to Q_1', in which the relation mapping h_1' is $\{Exon(E,L,T)\rightarrow Exon(Eid,Level,Tid)\}$ while the variable mapping ϕ_1' is $\{E\rightarrow Eid, L\rightarrow Level, T\rightarrow Tid\}$. M_1' is only valid for query tuples with $Eid<=4$ since 1) all the view tuples in V_1' satisfy $E<=4$; and 2) no lambda variables in V_1' and the head variables E,T in V_1' are mapped to head variables Eid,Tid in Q_1' respectively, which implies *visibility*. For the query tuples for which M_1' is a valid view mapping, one covering set can be constructed: $\{M_1'\}$. After the user selects the query subset or the entire query result, the *citation queries* associated with V_1' are executed and the *citation function* is applied to construct formatted citations.

3.3 The need for provenance

We now show how to reason about the *validity* of view mappings for query tuples in the context of aggregate queries and views via an example, and illustrate why RBM fails, motivating the need for *provenance*.

Example 2. Consider the following query and view:

$$\begin{array}{l}
\lambda T.V_2'(T,COUNT(L),SUM(L)):-Exon(E,L,T),\\
\qquad\qquad\qquad\qquad COUNT(*)<3\\
Q_2'(AVG(Level)):-Exon(Eid,Level,Tid), Tid<=4
\end{array}$$

Note that V_2' corresponds to an SQL query with a HAVING clause, since $COUNT(*)<3$ appears as a subgoal in the body. V_2' is parameterized by T, and is associated with a parameterized citation query CV_2' which retrieves the citation information for every transcript ID. First, we can derive a view mapping $M_2'=(h_2',\phi_2')$ where $h_2'=\{Exon(E,L,T)\rightarrow Exon(Eid,Level,Tid)\}$ and $\phi_2'=\{E\rightarrow Eid, L\rightarrow Level, T\rightarrow$

$Tid\}$. Due to the subgoal $COUNT(*) < 3$, V'_2 cannot be used to rewrite Q'_2 since not every tuple in $Q'_2(D)$ can be computed using tuples in $V'_2(D)$ for *every* database instance D. However, *some* query tuples may be still computable from view tuples when a database instance is given. Therefore, V'_2 is potentially useful for *fine-grained citations* for *some* tuples in the instance of Q'_2.

To see this, consider the instances of relations Exon, Gene and Transcript in Tables 1, 2 and 3 respectively. Given those instances, the corresponding instances of V'_2 and Q'_2 are shown in Tables 4 and 5 (ignore for now the last columns). Note that the citations for each view tuple retrieved by instantiated CV'_2 are also provided in Table 4 (see the column with green background).

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	3	2	e_3
t_{e4}	4	2	4	e_4
t_{e5}	5	3	5	e_5
t_{e6}	6	2	5	e_6
t_{e7}	7	2	5	e_7

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

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
t_{t4}	5	GK-207	rRNA	2	r_4

Table 4: Instance of view V'_2 with provenance and citation

	T	COUNT(L)	SUM(L)	citation	prov
$t_{v'_2 1}$	1	1	1	{Group: [‘Lee’]}	e_1
$t_{v'_2 2}$	2	2	6	{Group: [‘Joe’]}	$e_2 + e_3$
$t_{v'_2 3}$	4	1	2	{Group: [‘Liu’]}	e_4

Table 5: Instance of query Q'_2 with provenance

	AVG(Level)	prov
$t_{q'_2 1}$	2.25	$e_1 + e_2 + e_3 + e_4$

Although V'_2 and Q'_2 do not have the same aggregate terms, there exists a *computation rule* [15] from $COUNT$ and SUM to AVG: For the same input, the output of AVG can be computed using the output of $COUNT$ and SUM. By applying the computation rule over the aggregate terms $COUNT(L)$ and $SUM(L)$ under view mapping M'_2, $AVG(Level)$ in query tuple $t_{q'_2 1}$ is computable. This relies on the fact that: 1) under M'_2, V'_2 retains all the columns needed for the aggregation in Q'_2 (i.e. $Level$); 2) the predicates in V'_2 and Q'_2 retain the same set of base relation tuples used to construct the view tuple $t_{v'_2 1}, t_{v'_2 2}$ and $t_{v'_2 3}$ as well as the query tuple $t_{q'_2 1}$. Note that this implies *provenance.* Intuitively, the aggregate results in $t_{q'_2 1}$ can be obtained by further aggregation over $t_{v'_2 1} - t_{v'_2 3}$ in $V'_2(D)$. Meanwhile, citation of $t_{q'_2 1}$ can be produced by combining the citations of $t_{v'_2 1} - t_{v'_2 3}$, i.e. {`Group: [‘Lee’, ‘Joe’, ‘Liu’]`}.

However, it is not trivial to extend RBM to handle such cases. The core idea of RBM is to determine whether there exists *a* view tuple that can provide a citation to *a* given query tuple (before duplicates are removed), which is achieved by explicitly evaluating the predicates of every view in each individual query tuple. In contrast, every tuple in an aggregate query or aggregate view instance is derived from a *set* of tuples, which complicates the reasoning since we are looking for a group of view tuples that can match *exactly* a group of query tuples. We therefore adopt an alternative model, called the *Provenance-Based Model* (*PBM*).

4. PROVENANCE-BASED MODEL

We now present the model for determining valid view mappings for each query tuple using provenance. We start by introducing some basic concepts before presenting validity conditions for view mappings, first for conjunctive queries and then extending them to handle aggregate queries.

4.1 Basic concepts

To determine the validity of view mappings, it is necessary to understand the relationship between the schema of the query and the schema of the views under the view mappings. Validity also relies on the notions of 1) how-provenance polynomials; and 2) an isomorphism between how-provenance monomials and subgoals, to pave the way to reasoning about *valid view mappings* at the tuple level.

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 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 set of 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.

Example 3. Consider Q'_2 and V'_2 from Example 2 of Section 3.3. V'_2 has *finer granularity* than Q'_2 since V'_2 has one grouping variable T while Q'_2 does not have any grouping attribute. If we add variable Tid into the head of Q'_2, then V'_2 has the *same granularity* as Q'_2 since $\phi'_2(T) = Tid$.

How-provenance. We use the notion of *how-provenance* introduced in [24]. 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 $t_{q'_2 1}$ in $Q'_2(D)$ (see Table 5) is $e_1 + e_2 + e_3 + e_4$, which has four *how-provenance monomials.* It means that four tuples from base relation Exon with how-provenance tokens $e_1 - e_4$, respectively were used to create $t_{q'_2 1}$. 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.

Isomorphism between how-provenance monomials and subgoals. Since we need to reason about the *validity* of view mappings using how-provenance, we need to build connections between them. Such connections are natural since how-provenance tokens in the query tuple can be traced back to the corresponding base relation tuples. When the query is evaluated, those base relation tuples are *assigned* to relational subgoals which may be involved in view mappings. We therefore introduce the notion of *assignment*, before defining an *isomorphism* between how-provenance monomial and relational subgoals under an assignment.

As mentioned in [24], the how-provenance for a query or view tuple is a polynomial of how-provenance tokens. To simplify reasoning over these expressions, [6] defines a normal form for how-provenance polynomial: 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, which is closely connected to the notion of *assignment* as below.

Definition 1. **Assigment.** [6] An assignment γ for a (conjunctive or aggregate) query Q with respect to a database instance D is a mapping of the relational subgoals of Q to tuples in D that respects relation names, and induces a mapping over variables/constants. If a relational subgoal $R(x_1, \ldots, x_n)$ is mapped to a tuple $R(a_1, \ldots, a_n)$ then we say that x_i is mapped to a_i (denoted as $\gamma(x_1, \ldots, x_n) = (a_1, \ldots, a_n)$).

A tuple t in $Q(D)$ may have multiple *assignments*, (denoted as $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$), each of which should correspond to one *how-provenance monomial*. It is exemplified as below.

Example 4. Suppose we have a query which is a self join on relation Transcript, and retrieves some pairs of gene ids of the same type:

$$Q'_4(G, G') : -Transcript(T, N, Ty, G), T >= 4, Ty = Ty', \\ Transcript(T', N', Ty', G'), T' >= 4$$

The query result along with the how-provenance expression is shown in Table 6 using the instance of Transcript in Table 3. Note that the second and the third how-provenance monomial of the tuple $t_{q'_4 1}$ is written differently i.e. $r_3 * r_4$ vs $r_4 * r_3$ (although they are equivalent to each other), which represent different *assignments*. The former monomial $r_3 * r_4$ represents one *assignment* γ in which tuples t_{t3}=(4, HP-218, rRNA, 2) (with token r_3) and t_{t4}=(5, GK-207, rRNA, 2) (with token r_4) from Transcript are assigned to subgoals $Transcript(T, N, Ty, G)$ and $Transcript(T', N', Ty', G')$ respectively (denoted as $\gamma(T, N, Ty, G)$=(4, HP-218, rRNA, 2) and $\gamma(T', N', Ty', G')$=(5, GK-207, rRNA, 2)) while the latter one reverses the order of the assignment γ.

Besides, the first how-provenance monomial of $t_{q'_4 1}$ is written as $r_3 * r_3$ instead of the compact form (r_3^2). Furthermore, the coefficient of all the monomials in the how-provenance polynomial of $t_{q'_4 1}$ is 1 (although in a more compact form, $r_3 * r_4$ and $r_4 * r_3$ can be combined into $2 * r_3 * r_4$ or $2 * r_4 * r_3$)

Table 6: $Q'_4(D)$ along with how-provenance

	G	G'	prov
$t_{q'_4 1}$	2	2	$r_3 * r_3 + r_3 * r_4$ $+ r_4 * r_3 + r_4 * r_4$

For a given query tuple, [6] defines an *isomorphism* between *assignments* and the how-provenance monomials in a query. Borrowing some ideas from there, 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 before.

Definition 2. **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 $\gamma(B_i) = t_{bi} (i = 1, 2, \ldots, m)$ [7]. 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 4, consider the second how-provenance monomial $r_3 * r_4$ and corresponding *assignment* γ in query tuple $t_{q'_4 1}$ in Table 6. Since t_{t3} and t_{t4} are associated with how-provenance tokens r_3 and r_4 respectively, there should be an isomorphism F such that $F(Transcript(T, N, Ty, G)|\gamma)$ $= r_3$ while $F(Transcript(T', N', Ty', G')|\gamma) = r_4$.

4.2 Validity conditions without aggregation

We now discuss how to apply *provenance* to determine the validity of view mappings for conjunctive queries. Note that aggregate views have previously been shown to be *invalid* for rewriting conjunctive queries [34]. We therefore only consider view mappings of conjunctive views.

The validity conditions of view mappings can be divided into *schema-level conditions* and a *tuple-level condition*. A view mapping M is valid for a given query tuple iff M satisfies both *schema-level* and *tuple-level* conditions.

Definition 3. **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 distinguished variable $y \in \bar{Y}$ such that $\phi(y)$ is a distinguished variable; 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.

Definition 4. **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 5. Suppose Q'_2 and V'_2 in Example 2 are modified as follows by throwing away aggregate functions, and adding one predicate to V'_2:

$\lambda T.V''_2(T, L)$ $\quad : -Exon(E, L, T), E <= 3$

$Q''_2(Level)$ $\quad : -Exon(Eid, Level, Tid), Tid <= 4$

We can build the obvious view mapping $M''_2 = (h''_2, \phi''_2)$ from V''_2 to Q''_2. Using the instance of Exon in Table 1, the instances of V''_2 and Q''_2 can be constructed as in Tables 7-8.

Table 7: $V''_2(D)$ with how-provenance

	T	L	citation	prov
$t_{v''_2 1}$	1	1	{Group: ['Lee']}	e_1
$t_{v''_2 2}$	2	3	{Group: ['Joe']}	$e_2 + e_3$

Table 8: $Q''_2(D)$ with how-provenance

	Level	prov
$t_{q''_2 1}$	1	e_1
$t_{q''_2 2}$	3	$e_2 + e_3$
$t_{q''_2 3}$	2	e_4

We can show that M''_2 is a valid view mapping for the query tuple, $t_{q''_2 1}$ and $t_{q''_2 2}$, as follows: The *schema-level conditions* are satisfied because 1) the head variable L in V''_2 are mapped to the head variable $Level$ in Q''_2; and 2) the lambda variable T in V''_2 is mapped to Tid in the body of Q''_2.

The *tuple-level condition* also holds for the two result tuples. For example, for query tuple $t_{q''_2 2}$ (and view tuple $t_{v''_2 2}$), for its first monomials, the assignment and isomorphism under the assignment are γ (γ') and F (F') respectively. Since under the view mapping $M''_2 = (h''_2, \phi''_2)$, $h''_2(Exon(E, L, T)) = Exon(Eid, Level, Tid)$,
$F'(Exon(E, L, T)|\gamma') = e_2 = F(Exon(Eid, Level, Tid)|\gamma)$. So we say that M''_2 is a valid view mapping for the how-provenance monomial e_2 for query tuple $t_{q''_2 2}$. We can also prove that M''_2 is a valid view mapping for how-provenance monomial e_3 in tuple $t_{q''_2 2}$ and for e_1 in tuple $t_{q''_2 1}$.

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 function 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; this has been explored in previous work [15, 16] in the context of query rewriting using views with aggregation.

In particular, [15] 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 are *well-formed.* For example, the "mapper" function for AVG takes a set of rational numbers, $\{d_1, d_2, \ldots, d_k\}$, and maps each number d_i to a pair $(d_i, 1)$. The reduce function is pair-wise addition, whose result is 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. One of the most important properties of a well-formed aggregate function is *invertibility* [15]. An aggregate 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 6. Consider the following query and view:

$V'_6(E, T, SUM(L)) : -Exon(E, L, T)$
$Q'_6(E, SUM(L)) : -Exon(E, L, T)$

V'_6 computes a coarser-grained aggregation result than Q'_6 does. Both share the same aggregate function SUM, which is invertible. This means that we can take the sum of the aggregation results in V'_6 to get the result of Q'_6 under the obvious view mapping M'_6.

However, if we replace SUM with AVG, the aggregation result in V'_6 will not be useful to compute the aggregation result in Q'_6 under M'_6; the intermediate sum and count from V'_6 that were used in the terminating function (divide) cannot be regained to use in the further aggregation for Q'_6, 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 [15]. 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 [16], 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, as Example 2 shows, there is a computation rule from SUM and $COUNT$ to AVG, i.e. $SUM, COUNT \rightarrow AVG$. Such computation rules can be predefined by the DBAs.

The authors in [15] and [16] consider aggregate function requirements for potentially valid views to rewrite a query by combining the aggregate function properties mentioned above, which are adapted below for data citation:

Definition 5. **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 $M = (h, \phi)$ mapping V to Q:

1. V also has an aggregate function α with arguments Y, and $\phi(Y) = X$
2. there exists some *computation rule* $\beta_1, \beta_2, \ldots, \beta_m \to \alpha$ and $\beta_1, \beta_2, \ldots, \beta_m$ also appear (or can be derived by other computation rules) in the schema of V, all of which take same set of variables Y as arguments and $\phi(Y) = X$.
3. If Q has coarser granularity than V, 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.

Note that there is a special case in which the grouping variables of a view can be used to compute an aggregate term of a query under some view mapping, and in this case the view mapping is also potentially valid. In order to deal with this case, we assume that those grouping variables are associated with the identity function (a special aggregate function mapping its arguments to themselves), by which the rules in Definition 5 are then applicable.

Example 7. Suppose Q'_7 and V'_7 are defined as:

$V'_7(E, L, COUNT(*)) : -Exon(E, L, T)$
$Q'_7(E, AVG(L)) : -Exon(E, L, T)$

Although V'_7 has a finer granularity than Q'_7 under the obvious view mapping M'_7, there is no computation rule from $COUNT$ to AVG. However, it is possible to assign the identity function to the grouping variable L of V'_7 such that the following two computation rules work, and thus M'_7 satisfies the rules in Definition 5:

$IDENTITY, COUNT \to SUM$
$SUM, COUNT \to AVG$

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 6. **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 attributes* of Q, the following must hold:
   (a) If V is a *conjunctive* view, then for every grouping attribute X of Q there is an attribute Y in the head of V such that $\phi(Y) = X$.
   (b) If V is an *aggregate* view, then Q must have the same or coarser granularity than V 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 5.

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 7. **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 $\sum_{i=1}^{p} W'_i$ ($\sum_{i=1}^{p} W'_i$ again follows the normal form mentioned in Section 4.1) can be *mapped* to some monomial in W as a one-to-one function under M.

Example 8. Recall Q'_2, V'_2, and view mapping $M'_2 = (h'_2, \phi'_2)$ from Example 2. M'_2 is valid for query tuple $t_{q'_2 1}$. The reasons are as follows.
In terms of *schema-level conditions*, M'_2 is satisfied because 1) V'_2 has *finer granularity* than Q'_2 under M'_2; and 2) the arguments in the aggregate terms of V'_2, $COUNT(L)$ and $SUM(L)$, can be mapped to the aggregate term of Q'_2, $AVG(Level)$, and there exists a *computation rule*:
$COUNT, SUM \to AVG$.
The *tuple-level condition* also holds for the query tuple $t_{q'_2 1}$ if we compare its provenance to the sum of the provenance polynomials of the view tuple set $\{t_{v'_2 1}, t_{v'_2 2}, t_{v'_2 3}\}$, i.e. $W' = e_1 + e_2 + e_3 + e_4$, and the monomial mapping between W' and the how-provenance polynomial of $t_{q'_2 1}$ (i.e. $e_1 + e_2 + e_3 + e_4$) is a one-to-one function. This reasoning is more complicated than that in Example 5, 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 [35], *covering sets* are computed which cover as many *aggregate terms* in the query as possible using the fewest view mappings, i.e. that are *maximal and non-redundant.*

Example 9. Consider the following three views and query:

$V'_9(E, MAX(L)) : -Exon(E, L, T)$
$V''_9(E, MAX(T)) : -Exon(E, L, T)$
$V'''_9(E, MAX(T), MAX(L)) : -Exon(E, L, T)$
$Q'_9(E, MAX(L), MAX(T)) : -Exon(E, L, T)$

There is a valid view mapping from each individual view to Q'_9 for all query tuples, denoted M'_9, M''_9 and M'''_9 respectively, which form two *covering sets.* The first one is $\{M'_9, M''_9\}$, which *jointly* cover the two aggregate terms in Q'_9. The other covering set is $\{M'''_9\}$, which covers the same terms as $\{M'_9, M''_9\}$ does. The two covering sets provide two *alternative* ways to generate citations and are denoted $\{\{M'_9, M''_9\}, \{M'''_9\}\}$.

5. FROM THEORY INTO PRACTICE

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, which must be overcome to provide a solution with acceptable time performance. We discuss those challenges in this section, starting with a discussion of the algorithmic complexity of ProvCite, before discussing implementation details and optimizations used in the context of an example.

5.1 Algorithmic complexity

In what follows, we assume that the underlying database system is *provenance-enabled*, i.e. we do not consider the cost of carrying provenance through queries and generating the how-provenance of final results. Rather, we focus on the cost of *using* provenance to generate fine-grained citations.

An overview of our implementation is shown in Algorithm 1. The algorithm consists of three steps: 1) *Preprocessing*, 2) *Reasoning about valid view mappings*, and 3) *Covering sets calculation.*

The major overhead of the *Preprocessing* step is loading the query provenance into memory, which is determined by the underlying provenance-enabled database.

In the *Reasoning about valid view mappings* step, the time to check the validity of a view mapping includes: 1) the time to retrieve the view provenance from the database, which is proportional to the total number of how-provenance monomials in the view instance on average (denoted as N_{pv}); and 2) the time to compare the view provenance and query provenance in memory, which is proportional to the total number of how-provenance monomials in the query (N_{pq}). As a consequence, if there are m view mappings the time complexity for this step is $O(m*N_{pq})+O(m*N_{pv})$. Suppose k is the upper bound for the number of relational subgoals in the query or view body and the largest relation in the database has n tuples, then $O(N_{pq}) = O(N_{pv}) = O(n^k)$ and thus 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 not very large (usually no more than 1 million).

The time for the *Covering sets calculation* step depends on the *policies* on how to convert *covering sets* into formatted citations – see [35]. Since in the worst case the number of possible coverings sets may be exponential in m, if m is large and all covering sets are used in a policy this can be very costly. However, in practice m is small (e.g. 1 or 2) since views associate *different parts* of the database with citations and queries have a small number of subgoals. Performance is therefore acceptable even when all covering sets are used in a policy, as shown in Section 6. Furthermore, in practice we believe that a "minimal cost" policy will be used to generate concise citations, in which case covering sets are pruned, resulting in a cost which is linear in m.

5.2 Optimization and implementation (via an example

Our worst case complexity analysis shows that if naively implemented, it might be quite expensive to generate fine-grained citations, and therefore challenging to do with acceptable performance. We therefore use a number of optimizations, which are discussed below using an example which also illustrates how ProvCite is implemented.

Example 10. Given the views $V_1 - V_5$ defined in Section 3.1, suppose the query is as follows:

$$Q(T, N, COUNT(E), MAX(L)) : -Exon(E, L, T'), E <= 6 \quad Transcript(T, N, Ty, G), T = T'$$

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.

Next, all possible view mappings are constructed. We can find three view mappings, M_3, M_4 and M_5, under which the relational subgoals of V_3, V_4 and V_5 are mapped to subgoals of Q with the same name, respectively. Note that all the *schema-level conditions* are independent from individual query tuples, which can be applied to remove invalid view mappings early. In this example, M_3, M_4 and M_5 are all satisfied, since under each mapping all the grouping variables and at least one aggregate term of Q are *covered.*

In the *Reasoning about valid view mappings* step, since V_3 is a *conjunctive view*, the validity of its view mapping M_3 can be determined by reasoning about the *satisfiability of predicates* in V_3 under M_3 without retrieving the provenance of V_3. This is because the provenance of each tuple in the query result is expressed in terms of the base relation tuples, which can be evaluated against the view *predicates* to determine whether or not the tuple would appear in the view. For example, the validity of M_3 can be checked simply by examining the predicates of V_3. Since the first predicate, $T = T'$, in V_3 is the same as that in Q, every tuple in the query instance must satisfy it. The second predicate, $E >= 4$, is only related to relation Exon and the how-provenance tokens $e_1 - e_6$ in $Q(D)$. Looking at Table 1, only the tuples with how-provenance token $e_4 - e_6$ satisfy $E >= 4$. Thus M_3 is only valid for t_{q3} and t_{q4}, whose how-provenance polynomials only include $e_4 - e_6$ without any other how-provenance tokens from Exon.

In contrast, since both V_4 and V_5 are aggregate views, their how-provenance expressions are needed to check the *tuple-level conditions* of view mappings, which can be dealt with by two alternative strategies. One is to pre-compute their how-provenance (*eager strategy*) while the other one is to retrieve their how-provenance on the fly (*lazy strategy*). Their trade-offs will be discussed in Section 6.

The instances of V_4 and V_5, along with their how-provenance expressions, are presented in Tables 10 and 11, respectively. We find that view mapping M_4 is only valid for tuple t_{q3} since it shares the same set of how-provenance monomials with view tuple t_{v_41}.

Similarly, we can determine that M_5 is valid for query tuples t_{q1} - t_{q3}. Note that for tuple t_{q4}, although all of its how-provenance monomials exist in the view tuple t_{v_54}, it does not include $e_7 * r_4$ which is used to construct t_{v_54}, violating

```
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 valid view mapping step: Retrieve
  provenance of every view. For each query tuple $t$,
  determine valid view mappings by comparing the
  provenance of $Q$ and the provenance of $V$
3 Covering sets calculation step: Calculate covering sets
  by combining valid view mappings for each query tuple.
```

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

	T	N	COUNT(E)	MAX(L)	prov
t_{q1}	1	MB-203	1	1	$e_1 * r_1$
t_{q2}	2	PC-203	2	3	$e_2 * r_2 + e_3 * r_2$
t_{q3}	4	HP-218	1	2	$e_4 * r_3$
t_{q4}	5	GK-207	2	3	$e_5 * r_4 + e_6 * r_4$

Table 10: $V_4(D)$ with how-provenance polynomials

	T1	N1	COUNT(E)	prov
t_{v_41}	4	HP-218	1	$e_4 * r_3$
t_{v_42}	5	GK-207	2	$e_6 * r_4 + e_7 * r_4$

Table 11: $V_5(D)$ with how-provenance polynomials

	T1	N1	MAX(L)	prov
t_{v_51}	1	MB-203	1	$e_1 * r_1$
t_{v_52}	2	PC-203	3	$e_2 * r_2 + e_3 * r_2$
t_{v_53}	4	HP-218	2	$e_4 * r_3$
t_{v_54}	5	GK-207	3	$e_5 * r_4 + e_6 * r_4 + e_7 * r_4$

Table 12: $Q(D)$ with valid view mappings and covering sets (aggregate terms omitted)

	T	N	valid view mappings	covering sets
t_{q1}	1	MB-203	M_5	$\{\{M_5\}\}$
t_{q2}	3	PC-203	M_5	$\{\{M_5\}\}$
t_{q3}	2	HP-218	M_3, M_4, M_5	$\{\{M_3\}, \{M_4, M_5\}\}$
t_{q4}	3	GK-207	M_3	$\{\{M_3\}\}$

the *Tuple-level condition.* Intuitively, since the max term may come from this component of the monomial ($e_7 * r_4$), t_{v_54} should not provide citation information for t_{q4}.

Finally, valid view mappings and derived covering sets are shown in Table 12. Note that for tuple t_{q3}, there are two covering sets, $\{M_3\}$ and $\{M_4, M_5\}$; other combinations of view mappings either redundantly or non-maximally cover the query's aggregate terms, and therefore are not valid.

Three strategies are applied to speed up the covering sets calculation: 1) representing coverings sets using bit arrays; 2) applying clustering algorithms to avoid an explosion of intermediate results; and 3) query tuples associated with the same set of valid view mappings form a *reasoning group* to avoid repetitive computations of covering sets. For example, t_{q1} and t_{q2} have the same set of valid view mappings, $\{M_5\}$, which should end up with the same covering sets. So those two tuples are grouped together to compute covering sets once. These optimizations can result in orders of magnitude speed-up.

6. EXPERIMENTS

6.1 Experimental set-up

We implemented ProvCite 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 central memory.

Datasets. Two realistic datasets were used in addition to GENCODE: Hetionet[1] and DBLP-NSF[2]. Summary information of the three datasets, including the number of relations, average size per relation, and the size of the largest relation is presented in Table 13.

Table 13: 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

We converted Hetionet, which is stored in Neo4j, into a relational database[3]. DBLP-NSF was developed by the authors of [35], and integrates DBLP publication information with NSF award information to augment traditional paper citations with funding information.

Workloads. We test the performance of ProvCite using both *synthetic* and *realistic* workloads. As mentioned earlier, one essential step for aggregate views is to compare the provenance of views and queries. In order to retrieve the provenance of views, we can use either *eager* or *lazy* strategy. We compare these two strategies using both forms of workloads. The performance also depends on the *policies.* As mentioned in Section 5, different policies can lead to different results, and can generate either all or some of the covering sets. Due to space limitations, only the case where the 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 [35] measuring the *total reasoning time* to generate the covering sets (t_{cs}) and the *citation generation time* after covering sets are constructed (t_{cg}). The *citation generation time* (t_{cg}) 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.

In [35], 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); 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 is also considered in the experiments.

The trade-offs between ProvCite and two implementations of the RBM proposed in [35], 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. Complete details of the views and queries used are available in our Github repository[4].

[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

Table 14: Notation used in the experiments

Notation	Meaning
t_{cs}	total reasoning time to generate the covering sets for all query tuples
t_{cg}	citation generation time after covering sets are constructed
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

To mimic the summary information provided by GENCODE, we defined aggregate views to compute the total number of transcripts per gene, and 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 [35]. 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 total number of grants per institution[6]. Some realistic aggregate queries are designed to represent other summary information, such as total number of publications per institute ($q3$) 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 14 provides a summary of the notation defined 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 size of provenance in both the query and view instances on time performance (Exp1), and the relative performance of ProvCite and two implementations of the RBM, i.e. TLA and SSLA while varying the number of view mappings (Exp2) and number of view predicates (Exp3).

Exp1. This experiment measures how the total reasoning time t_{cs} 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 10, which is a reasonable number in practice. Both N_{pq} and N_{pv} are varied from 50K to 1M. t_{cs} is measured for different (N_{pq}, N_{pv}) pairs under both the *eager* and *lazy* strategy.

Results. The time performance, t_{cs}, is shown using 3D surfaces in Figure 4a, with the eager strategy shown in blue and the lazy strategy shown in red. It shows that the time performance under the eager strategy is only slightly better than under the lazy strategy (10% - 40% faster). In the worst case, in which both N_{pv} and N_{pq} are 1 million, it takes no more than 150 seconds. This case will be rare, so the performance in practice should be much better. We also measured the extra space needed for the eager strategy, where the provenance of views is precomputed and stored in the database; it takes about 180 MB in the database to store the provenance for each view when the instance of the view includes up to 1 million how-provenance monomials.

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. So there is no difference between *eager* and *lazy* strategy since the provenance of views is not necessary. The query is a fixed aggregate query, with 500k 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.

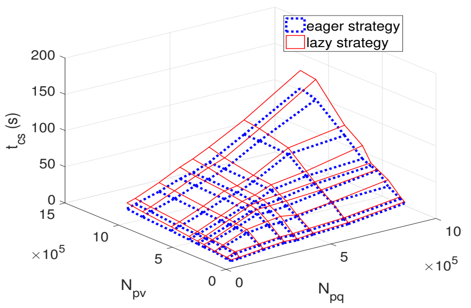
Results. The experimental results are presented in Figure 4b, which shows that t_{cs} grows rapidly with the increasing number of view mappings (N_v) for all the three approaches. This is due to the fact that an exponential number of covering sets are generated, which is inevitable. It is worth noting that ProvCite *outperforms* TLA when $N_v < 40$, and is on par with SSLA until $N_v > 30$. Furthermore it only adds a little extra overhead when $N_v > 40$ compared to the competitors.

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 500k tuples. The number of view mappings is fixed at 10 and there are initially no predicates. At each run, one more local predicate is added. As shown in [35], increasing N_p can significantly influence the performance of TLA and SSLA by 1) increasing the query time since the query is extended to explicitly evaluate the satisfiability of predicates of the views in the query instance; 2) increasing the number of *reasoning groups* since we need to compute covering sets for each group and thus more *reasoning groups* in the query instance means more reasoning time. In theory, ProvCite will also suffer from a large number of groups but will save on query time.

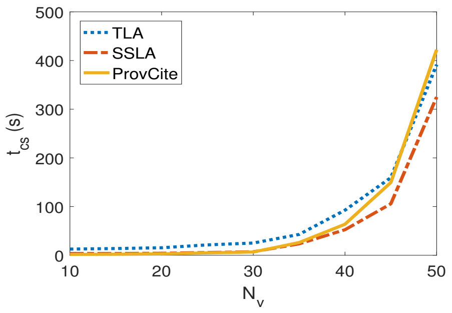
Results. The experimental results are shown in Figure 4c, which matches the analysis above. As the number of predicates increases, t_{cs} increases slowly for ProvCite. In contrast, TLA and SSLA are twice as slow as ProvCite for large N_p. To understand the reason for this, the query time for TLA and SSLA is also presented in this figure, which implies that the increasing query time becomes the major

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

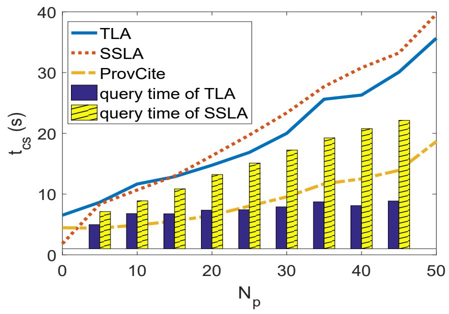
[6]`https://dellweb.bfa.nsf.gov/awdlst2/default.asp`



(a) t_{cs} with varied N_{pq} and N_{pv}



(b) t_{cs} with varied N_v



(c) t_{cs} with varied N_p

Figure 4: Experimental results for synthetic workloads

overhead for both TLA and SSLA, thus slowing down the computation.

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), 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, bad performance is unavoidable. However these cases are unlikely to happen in practice. We therefore expect reasonable time performance for realistic workloads.

In the case of aggregate views where how-provenance is necessary, the eager strategy beats the lazy strategy in terms of time. The speedup is small (10%-40%), however, extra space is needed (up to 180 MB per view). So in practice, the choice between the eager or lazy strategy depends on whether speed or space is more important. In comparison to the previous approaches, TLA and SSLA, ProvCite is not only more powerful in that it supports aggregate views but is also (surprisingly) frequently more efficient.

6.2.2 *Realistic workloads*

The experimental results for realistic workloads are presented in Table 15, which includes the time to generate covering sets (t_{cs}) for both the lazy and eager strategies, 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 query time to simply generate instance. Except for $q1$, most of t_{cs} is less than 10 seconds for all queries. Although N_{pq} is more than one million in $q1$, the total reasoning time (t_{cs}) is only about a half minute under both strategies, which is an acceptable considering the large query instance. To see how this compares to simply generating the query output, we list the query running time in the last column. When users are browsing the query result, the system can generate covering sets for all query tuples in the back-end, ready to instantly construct formatted citations when users select tuples of interest.

Discussion. The experimental results above 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

Table 15: Experimental results on realistic datasets

Query	t_{cs} (s) (eager)	t_{cs} (s) (lazy)	N_{pq}	N_v	N_p	query time (s)
$q1$	22.51s	36.84s	1237914	1	0	1.02
$q2$	4.40s	6.03s	203835	2	0	0.13
$q3$	8.47s	11.23s	507515	2	0	1.15
$q4$	3.72s	5.68s	416716	1	0	0.67
$q5$	4.56s	6.82s	243901	3	0	0.51

instance (N_{pq}) is more than 1 million (as in $q1$), the reasoning time (t_{cs}) can be large (up to 150 seconds). However, the time shown for $q1$ is significantly smaller since the number of view mappings (N_v) is only 1, and the reasoning time relies on both N_v and N_{pq}.

7. CONCLUSIONS

This paper builds 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.

The Provenance-Based Model was implemented in ProvCite, and extensive experiments conducted under both *synthetic* and *realistic workloads*. The results show that ProvCite can not only handle a larger class of queries than pure Rewriting-based approaches (which assume conjunctive queries and views, e.g. [35]), but is much faster in some cases. However, the approach assumes a *provenance-enabled DBMS*. Trade-offs between an *eager* versus *lazy* strategy for generating view provenance was also explored. The choice involves a trade-off between speed and space.

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 use citation within machine learning pipelines.

8. ACKNOWLEDGMENTS

This research was partially funded by NSF IIS 1302212 and NSF ACI 154736. The authors would like to thank Peter Buneman for conceiving the idea that data citation is a computational challenge and recognizing its connection to provenance; to Val Tannen for discussions on query rewriting and provenance; and Boris Glavic for his support in understanding details of GProM's source code.

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. B. Davidson, W. Hu, and Y. Wu. Automating data citation in CiteDB. *Proceedings of the VLDB Endowment*, 10(12):1881–1884, 2017.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

[34] 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.

[35] 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.

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

[37] 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.