A Coq Formalization of Data Provenance

VÉRONIQUE BENZAKEN, SARAH COHEN-BOULAKIA, ÉVELYNE CONTEJEAN, CHANTAL KELLER, and RÉBECCA ZUCCHINI, LRI, Université Paris Saclay and CNRS (UMR8623), France

In multiple domains, large amounts of data are daily generated and combined to be analyzed. The interpretation of these analyses requires to track back the *provenance* of combined data with respect to initial, raw data. The correctness of the provenance is crucial in many critical domains, such as medicine to prescribe treatments. In this article, we propose the first provenance-aware extended relational algebra formalized in a proof assistant (Coq), for a non trivial subset of database queries: queries containing aggregates, null values, and correlated sub-queries. The formalization is validated by an adequacy proof with respect to standard evaluation of queries. This development is a first step towards *a posteriori* certification of provenance for data manipulation, with strong guaranties.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Correctness**; *Software verification*; • **Theory of computation** → Logic and verification; Type theory; **Program verification**; **Program semantics**; **Data provenance**.

Additional Key Words and Phrases: extended relational algebra, Coq

ACM Reference Format:
Véronique Benzaken, Sarah Cohen-Boulakia, Évelyne Contejean, Chantal Keller, and Rébecca Zucchini. 2021. A Coq Formalization of Data Provenance. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3437992.3439920

1 INTRODUCTION

In the medical field, the choice of a treatment or protocol has been increasingly dependent on data analysis. For example, sequencing methods are a classic application of data analysis in cancerology: it is used to estimate the origin of a cancer, the mutations of a cell, the type of evolution of a cancer, as well as which treatment would be more suited for a particular cancer. These analyses need a large amount of data to be more efficient and fortunately, in the last decade, the development of new tools, in particular in sequencing, has led to a massive influx of raw data. However, it has become a challenge to perform complex analyses of the ever-growing amount of data without introducing bias. Moreover, the sequence of several processes communicating intermediate data as well as final data makes it difficult to understand the results of these analyses and their limitations, while also increasing the risk of error propagation. These questions are not specific to the medical domain, but a central point in every domain that requires large data analysis, such as IoT, space probes, … Such data analyses often involve many data management systems and are called *scientific workflows*.

To cope with these issues, the database community proposed a formalism called *provenance*. The role is to keep track of data all along their transformation from raw data to final analysis: why do we obtain some data in the result (*why-provenance*) or not (*why-not-provenance*), where does it come from (*where-provenance*), etc. Most of these provenance information are computed by the means of annotations on data, as presented in the example below.

Scientific workflows compute these annotations and rely on them to ensure provenance of data and what it entails, such as result understanding and reproducibility of analyses. Having strong guaranties on annotations is thus crucial in critical domains of application of scientific workflows, especially as it allows producing FAIR data [11].

This article is a step towards such a goal: we study and formalize in the Coq proof assistant such annotations in the case of the relational algebra, which is a central language for relational databases, the most common used database management systems (through the SQL language). This work thus provides certified provenance for relational databases, but also paves the way towards a framework to certify provenance in more complex workflows where many database systems interact.

1.1 Example

To explain the concept of provenance in the case of relational databases, we provide the example of the study of two medical treatments for breast cancer (*e.g.*, hormonal treatment or chemotherapy).

Relational databases are made up of *relations*, two examples of which are on **Table 1** and **2**. Let us first look at the four central columns (called *attributes*): for each patient designated by its **Name**, we indicate if s/he had a **Rem**ission of the cancer, after receiving **Tre**atment A or B at some **S**tage of the illness. The elements of a relation (the lines of the table) are called *tuples*; here, we refer to them using the names of the first column, t_1 to $_4$ for the first relation and t_5 to t_7 for the second one. It enjoys a bag semantics: the same tuple can be present more than once. We will explain later the role of the last column.

These two relations form our raw data: they are observations gathered by two doctors (respectively in relations r_1 and r_2) treating these patients. Note that among the patients there are two who are both named Jones and two who are both named Smith.

Now, we analyze these data: we want to know if treatment A is more efficient than treatment B in the early stage of this cancer. To this end, we first gather all the observations by computing the *union* of the two relations, whose result is presented in **Table 3**. After that we need to *select* only our data of interest so we keep the patients with a stage under 2 and who are in remission (**Table 4**). Then we do not need the remission and stage columns any more, so we can remove them by *projecting* on the other ones (**Table 5**). Finally, we obtain the number of patients that are in remission depending of the treatment, by *grouping* the patients depending on the treatment and *aggregating* the result by the number of tuples in each group: in **Table 6**, we have three patients that took treatment A (Smith twice and Johnson once), and only one that took treatment B (Miller).

In the syntax of *relational algebra*, this computation is written by the following *query*:

$$\Gamma_{treatment|count}(\pi_{name,treatment}(\sigma_{s\leqslant 2\wedge rem=Y}(r_1 \cup r_2)))$$

In this query, $\cup$ computes the union of the two initial relations, σ is the selection operator, π the projection, and Γ performs both grouping then aggregation on the groups.

To keep track of provenance, the mechanism we study in this article is to annotate tuples, and evaluate queries on these annotations rather than the content of the tuples, in a process similar to abstract interpretation. On our example, it corresponds to the last column **Annot** of each table.

Starting from initial annotations on raw data, to each operator of the relational algebra corresponds an operation on the annotations: the union adds the annotations, the selection rules out some of them, projection collapses tuples and thus also adds annotations. Aggregates behave a bit differently: the result of the query does not depend only on values, but also on annotations (*e.g.* $1 \circledast (a + d) \oplus 1 \circledast g$). The intuition is that the result of an aggregate may depend on

Table 1. relation r_1

	Name	**Rem**	**Tre**	**S**	**Annot**
t_1	Smith	Y	A	1	a
t_2	Jones	N	B	2	b
t_3	Garcia	N	A	4	c
t_4	Smith	Y	A	2	d

Table 2. relation r_2

	Name	**Rem**	**Tre**	**S**	**Annot**
t_5	Miller	Y	B	2	e
t_6	Jones	N	B	2	f
t_7	Johnson	Y	A	2	g

Table 3. union of relations r_1 and r_2

	Name	**Rem**	**Tre**	**S**	**Annot**
t_8	Smith	Y	A	1	a
t_9	Jones	N	B	2	b + f
t_{10}	Garcia	N	A	4	c
t_{11}	Smith	Y	A	2	d
t_{12}	Miller	Y	B	2	e
t_{13}	Johnson	Y	A	2	g

Table 4. selection of stage $\leqslant$ 2 and remission = Yes

	Name	**Rem**	**Tre**	**S**	**Annot**
t_{14}	Smith	Y	A	1	a
t_{15}	Smith	Y	A	2	d
t_{16}	Miller	Y	B	2	e
t_{17}	Johnson	Y	A	2	g

Table 5. projection on name and treatment

	Name	**Tre**	**Annot**
t_{18}	Smith	A	a + d
t_{19}	Miller	B	e
t_{20}	Johnson	A	g

Table 6. count, grouping by treatment

	Tre	**count**	**acount**	**Annot**
t_{21}	A	3	$1 \circledast (a+d) \oplus 1 \circledast g$	δ (a+d+g)
t_{22}	B	1	$1 \circledast e$	δ (e)

the weight of each aggregated tuple (see next paragraph). The resulting tuple is itself annotated by a combination of annotations (*e.g.* δ (a+d+g)). We will detail these aspects in Sec. 2.3.

As in abstract interpretation, the domain of the annotations has a direct influence on what we compute. The most simple domain is natural numbers, to represent throughout the query the number of occurrences of tuples in a table. On our example, we thus start with each tuple being present once: $a = b = c = d = e = f = g = 1$. Then, union adds the number of occurrences (bag semantics)[1], and similarly when projection collapses tuples. Finally, when counting, we have to take these numbers of occurrences into account, which showcases the fact that the resulting value depends not only on initial values but also on annotations. The annotation of the resulting tuple can be either one or zero: the number of occurrences of the tuple is one if it is actually present, *i.e.* if at least one of the aggregated tuples has a non-zero annotation, and zero otherwise. This domain of natural numbers thus matches the traditional bag semantics of relational algebra, as we will show in our main theorem of the paper.

Other domains reflect other provenance properties: annotations generalize the bag semantics. We will detail some common domains in Sec. 2. An important property to notice is that by keeping the domain abstract, we compute equations, such that the variables appearing in an equation exactly reflects from which tuples of the initial data the generated tuple comes from. For instance, in **Table 3**, the annotation b+f reflects the fact that tuple t_9 comes from two tuples, whereas the other tuples come from only once. These universal annotations, based on polynomials as a domain, will be explained in Sec. 2 as well.

1.2 Contributions and Outline

This example illustrates the concept of provenance in relational databases. In this article, we provide strong guaranties for provenance in this setting, by formalizing it, in Coq, for an extension of relational algebra.

Our work is articulated as follows. Our first contribution is a formal and generic semantics for a provenance-aware relational algebra, extended with a subset of aggregates (Sec. 4). We validate our semantics by showing its adequacy with the standard relational algebra in the case where annotations are natural integers (Sec. 4.4). Finally, we propose to instantiate our generic semantics to offer an executable interpretor for provenance in relational algebra with several annotations used in practice (Sec. 5).

Our work builds upon the Datacert library (Sec. 3), which formalizes relational algebra in Coq. We will start with detailing the syntax and semantics of provenance as presented in the literature (Sec. 2).

Figure 1 illustrates our contribution (black and red) and the building blocks we build upon (grey). The formal development as well as examples can be found at https://framagit.org/formaldata/provcert.

2 PROVENANCE IN DATABASES

2.1 Context

As presented in the example, a usual way to track how operators affect data when processing a query is to expand databases with annotations. It manifests by adding a new column (in our examples **Annot**) that associates an annotation to each tuple. To observe the impact of running a query on the annotations, we propose to focus on **Table 3** that represents the union between relations of **Tables 1** and **2**. It contains all the tuples that appear at least in one of the initial relations.

In the case of annotations representing occurrences of tuples, the annotation of a final tuple (for example t_8) is equal to the sum of occurrences (for t_8, $b + f$) of this tuple in the initial relations (disjoint union). For a tuple that appear only in one of the initial relations (like t_1), the annotation of the final tuple is $a = a + 0$: tuples that do not appear in one of

[1]It corresponds to `union all` in SQL.

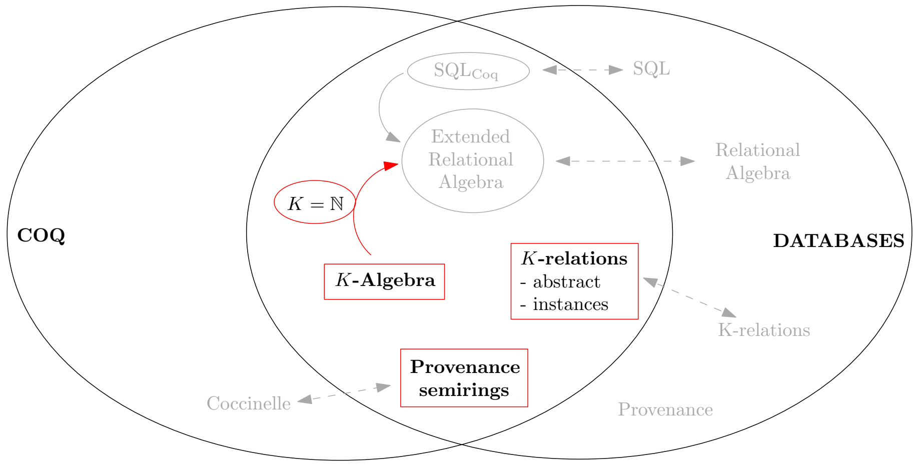

Fig. 1. Contributions

the relations can be considered as having zero as an annotation. Thus, we notice that the union of two tuples requires the an operation "+" and a zero on annotations.

More generally, for any domain of annotations, the union operator will combine annotations by the mean of + and zero of some algebraic structure. As the other tables of the example show, to each operator of the relational algebra is associated an operation on annotations. To reflect the associativity, commutativity and distributivity of the operators of databases, the operations on the annotations need to verify those properties. These characteristics suggest that annotations are at least in a commutative monoid. In practice, to capture the richness of relational algebra, the annotations are put in a unifying algebraic structure: the commutative semirings [14].

We have seen on **Table 6** that aggregates combine values with annotations. This combination requires two operations (that depend on the aggregate): an addition on values $\oplus$, and a tensor product between values and annotations $\circledast$. Again, we can remain abstract: these operations are fully captured by equipping the domain of values with a semimodule over the semiring of annotations. The annotations themselves are combinations of annotations with some function δ, that depends on the semiring of annotations. To sum up, to treat some aggregate, one has to extend the semiring of annotations K with a δ function computing one element of K from a set of elements of K, and to equip the values with a K-semimodule for each considered aggregate.

A relation is usually represented by a bag of tuples. A more abstract way to represent a bag is as a function from tuples to natural integers (number of occurrences). A *K-relation* [14] generalizes this idea: a function from tuples to some commutative semiring K, representing the domain of provenance annotations.

2.2 The Extended Relational Algebra

We now present the syntax of our database language: the positive relational algebra, extended with a restricted subset of aggregates.

Definition 1 (Syntax of queries)**.**

$$q' ::= r \mid q' \cup q' \mid \sigma_f(q') \mid \pi_s(q') \mid \epsilon_{<>} \mid q' \bowtie q'$$

$$q := q' \mid \Gamma_{gb|a}(q')$$

First, we present quickly the traditional semantics of the operators of the algebra (without annotations).

The first four operators of the relational algebra q' were used in the example of the introduction. The first one refers to a predefined relation r (such as r_1 and r_2 in the example). The operator $\cup$ represents the disjoint union. The operator σ performs selection: only the tuples satisfying a first-order logical formula f are kept. The operator π corresponds to the projection on a subset s of the attributes of q': we keep only the columns whose names are in s.

The next operator is $\epsilon_{<>}$: it is an attributeless relation only containing the empty tuple[2]. Finally, the $\bowtie$ operator represents the natural join. An example of this operator is given in **Tables 7** and **8** (we do not consider the last column at this stage): it combines the tuples of two relations when they are equal on their common attributes.

Table 7. relation r_3

	Name	**City**	**Annot**
s_1	Smith	Cambridge	h
s_2	Jones	New York	i
s_3	Bridges	Roma	j
s_4	Garcia	Portland	k

Table 8. natural join of r_1 and r_3

	Name	**Rem**	**Tre**	**S**	**City**	**Annot**
s_5	Smith	Y	A	1	Cambridge	a * h
s_6	Jones	N	B	2	New York	b * i
s_7	Garcia	N	A	4	Portland	c * k
s_8	Smith	Y	A	2	Cambridge	d * h

We work on an extension of this relational algebra q with the top-level aggregation operator Γ[3]. It first groups the tuples of q' on the expressions gb, then applies the aggregate function a on each group. An example was given in **Table 6**: the operator first groups the tuples in two different groups depending on the treatment (treatment A or B), then it applies the *count* aggregate on each group, which counts how many tuples are present in each group. Other common aggregate functions are $sum(x)$ (resp. $avg(x)$), summing (resp. computing the average of) the values of an integer attribute x, or $max(x)$, computing the maximum value of an integer attribute x.

2.3 Annotations

We now present the semantics of our algebra with annotations in a semiring $(K, 0, 1, +, *)$. We first explain the main intuitions, before giving the mathematical definitions.

As r is a predefined relation, the annotations of each tuple are also predefined. The semantics of the empty tuple $\epsilon_{<>}$ is very straightforward: it is a K-relation that associates 1 to the empty tuple and 0 to the others. As we have seen with the example, the union of two queries is the K-relation that sums the annotations of each initial K-relation.

We now focus on the case of the selection by a formula f. In the example (**Table 4**), the selection formula corresponds to "the stage has to be under 2 and the patient is on remission". The tuples that satisfy this formula are kept as well

[2]It is mainly used as a neutral element for the $\bowtie$ operator.
[3]Compared to SQL, we restrict the use of the aggregate to be the last operator applied, and the aggregate function can only be applied to one attribute. Moreover, we do not consider `having`. We leave these generalizations for future work.

as their annotation (like the tuples t_8, t_{11}, t_{12} and t_{13}). For the other tuples (like t_9 and t_{10}), they do not appear in the result: their annotations are then 0.

Concerning the case of the projection, we can look at **Table 5** that represents the projection on the attributes **Name** and **Tre**. We particularly focus on the tuple t_{18}: it is the result of the projection of t_{14} and t_{15}. So the annotation of t_{18} is the addition of the annotations of tuples whose projection is equal to t_{18}. More generally, the K-relation for projection is the function from a tuple t to the sum of annotations of the projected tuple equal to t.

Again, natural join is illustrated by **Table 8**. Since it combines tuples, the result is the K-relation that multiplies the annotations of each initial K-relation.

To picture the case of aggregates, we focus on the same tuple t_{21} as before. In the case of natural numbers, the annotation of this tuple depends whether it appears or not in the result: it is 0 if a, d and g are all equal to 0, and 1 otherwise. The expression $\delta(a + d + g)$ exactly computes it. More generally, the resulting tuple associated to a group X has for annotation $\delta_K(a_1 + .. + a_n)$ where the a_is are the annotations of the initial tuples of X. δ_K depends on the semiring K; the main property it must verify is $\delta_K(0) = 0$.

Associating annotations in K to tuples of the result of operators is called a *K-algebra* [14]. It is mathematically defined as follows.

Definition 2 (Semantics of the K-algebra)**.**

- $[\![r]\!] = t \mapsto \texttt{annot (r, t)}$
  where *annot* is the function that associates an annotation to each tuple t of the relation r
- $[\![\epsilon_{<>}]\!] = t \mapsto \begin{cases} 1 & \text{if } t = <> \\ 0 & \text{otherwise} \end{cases}$
- $[\![q'_1 \cup q'_2]\!] = t \mapsto [\![q'_1]\!](t) + [\![q'_2]\!](t)$
- $[\![\sigma_f(q')]\!] = t \mapsto \begin{cases} [\![q']\!](t) & \text{if } eval(f, t) = \top \\ 0 & \text{otherwise} \end{cases}$
  where *eval* evaluates the logical formula f on the tuple t
- $[\![\pi_s(q')]\!] = t \mapsto \sum_{t' \text{s.t.} proj_s(t) = proj_s(t') \wedge [\![q']\!](t') \neq 0} [\![q']\!](t')$
  where $proj_s(t)$ is the projection on the set of attributes s of the tuple t
- $[\![q'_1 \bowtie q'_2]\!] = t \mapsto [\![q'_1]\!](t_1) * [\![q'_2]\!](t_2)$
  where t_i corresponds to the projections of t on the set of attributes associated with q_i
- $[\![\Gamma_{gb|a}(q')]\!] = t \mapsto \begin{cases} \delta_K(\sum_i a_i) & \text{if } t = (v_1 \circledast_a a_1) \oplus_a \dots \\ & \qquad \oplus_a (v_n \circledast_a a_n) \\ 0 & \text{otherwise} \end{cases}$
  where $v_1, \dots, v_n$ are the values appearing in the tuples of the group gb over the query q', and $a_1, \dots, a_n$ are their annotations.

For this definition to be valid, the sum for the projection operator π must be finite. It is indeed the case, since annotations must be 0 for all tuples except a finite number of them (we require this property for the base case of a predefined relation, and it is propagated by the definitions). It reflects the standard finiteness property of the relational model.

2.4 Domains of Annotations

We have already seen that using the set of natural numbers as the semiring K leads to annotations computing the number of occurrences of tuples, that is to say the bag semantics of SQL. Other commonly used domains are the following.

- The ring of Booleans ($\mathbb{B}$, $\bot$, $\top$, $\vee$, $\wedge$) simply indicates the presence or absence of a tuple (allowing one to recover the set semantics of SQL).
- It can be abstracted by the semiring of Boolean functions which compute not only one possible set of tuples, but all the possible sets (called *possibles worlds*) and can be useful to efficiently represent different instances of some database [14].
- The *security* semiring ($\mathbb{S}$, $0_\mathbb{S}$, $1_\mathbb{S}$, min, max) where $\mathbb{S}$ is an ordered set containing security levels (such as $0_\mathbb{S}$ being "not secure", then "confidential", "secret", and $1_\mathbb{S}$ being "top secret") computes the security level of the resulting data.
- The semiring of tropical integers ($\mathbb{N}^\infty$, 0, 1, +, *) computes the (possibly infinite) cost to obtain some data.

As there exists numerous domains for annotations, changing the domain would require to compute again the provenance for the same set of queries and databases, which is time- and energy-consuming. In the case without aggregates [14], people proposed a framework to encompass all the possible semirings: the positive algebra provenance semiring, consisting of polynomials with coefficients in $\mathbb{N}$.

Definition 3 (Positive algebra provenance semiring)**.** Let X be the set of tuple ids of a database instance I. The positive algebra provenance semiring is the semiring $(\mathbb{N}[X], 0, 1, +, .)$ of polynomials with interderminates from X and coefficients from $\mathbb{N}$ with + and . operators.

As explained at the end of Sec. 1.1, this semiring computes generalized annotations for a query. First, to obtain the annotations in the desired semiring, one only has to evaluate the resulting polynomial by replacing monomials with initial annotations, and substituting the addition and multiplication of polynomials by those of the semiring. Second, more abstractly, the monomials appearing in the resulting annotation of a tuple precisely indicate from which original tuples it comes from.

3 PRESENTATION OF THE DATACERT LIBRARY

We now present the main building block we base our formalization on.

The Datacert library [3] provides the most complete mechanized semantics for SQL queries with aggregates, NULL values and correlated queries, and formally relates them to an extension of relational algebra. In this work, we extend the relational algebra part with provenance. We also use Datacert as a reference to validate our semantics based on generic K-relations: when K is instantiated with natural numbers, we can prove that the two semantics coincide (Sec. 4.4). In addition, this semantics is executable (both in Coq and after extraction), which allows us to actually compute provenance.

The Datacert formalization is divided into two layers:

- an abstract layer provides the syntax and semantics for SQL and algebra queries, with an abstract data model, abstract types for attributes, values, tuples, predicate symbols, … making it easily extensible and abstracting away implementation details (Sec. 3.1);
- a proof of concept instantiates these types with standard values, to provide the executable semantics (Sec. 3.2).

3.1 Syntax and Semantics of Extended Relational Algebra

As our work is restricted to a subset of the queries, we present only the associated fragment of the syntax proposed in Datacert.

The query syntax is defined by a recursive type `query`:

```
Inductive query : Type :=
  | Q_Table : relname ⟶ query (*Relation identified by a name*)
  | Q_Union : query ⟶ query ⟶ query (*Union*)
  | Q_Sigma : formula ⟶ query ⟶ query (*Selection*)
  | Q_Pi : select_list ⟶ query ⟶ query (*Projection*)
  | Q_Empty_Tuple : query (*Relation containing the empty tuple*)
  | Q_NaturalJoin : query ⟶ query ⟶ query (*Natural join*)
  | Q_Gamma : select_list ⟶ list aggterm ⟶ query ⟶ query. (* Aggregates *)
```

We now detail the different types used to define `query`. The first type is `relname`, that is an abstract type for relation names. As explained above, the proof of concept (Sec. 3.2) instantiates this type by the type `string`. The type `formula` is a recursive type for SQL first order logical formulas, that we do no detail here (we refer to [3]). The type `select_list` occurs in projections and aggregates. It consists in a list of couples of type `aggterm * attribute`, where `aggterm` represents SQL terms with function and/or aggregate symbols (such as `a + b` or `sum(x)`) and `attribute` is an attribute name given to the term (standing for the `as` construct). For instance, the projection $\pi_{name,treatment}(q_1)$ in the example corresponds to `Q_Pi [(name,name);(treatment,treatment)] q`$_1$, and the aggregate $\Gamma_{treatment|count}(q_2)$ corresponds to `Q_Gamma [(count(*), count)] treatment q`$_2$.

This definition matches our grammar in **Definition 1**, except that it does not enforce the aggregate to be the last applied operation, and to have only one aggregate. To do so, we have defined a predicate `well_formed` that takes a Boolean and a query and checks that this property is verified. The Boolean argument is a flag to indicate whether or not we are in the first call of the function.

The semantics is defined by a recursive function which associates a finite bag of tuples to each query. It is based upon the library of finite bags called `Fbag`, which is part of an extension of the Coccinelle library [9].

```
Fixpoint eval_query env q : bagT :=
  match q with
  | Q_Table r ⇒ instance r
  | Q_Union q1 q2 ⇒
    if sort q1 =? sort q2
    then Fbag.union (eval_query env q1) (eval_query env q2)
    else Fbag.empty
  | Q_Sigma f q ⇒
    Fbag.filter
      (fun t ⇒ is_true (eval_sql_formula (env_t env t) f))
      (eval_query env q)
  | Q_Pi s q ⇒ Fbag.map (fun t ⇒ projection (env_t env t) s) (eval_query env q)
  | Q_Empty_Tuple ⇒ Fbag.singleton empty_tuple
  | Q_NaturalJoin q1 q2 ⇒
    natural_join_bag (eval_query env q1) (eval_query env q2)
  | Q_Gamma s lf f q ⇒
  Fbag.mk_bag
    (map (fun l ⇒ projection (env_g env (Group_By lf) l) s)
    (make_groups env (eval_query env q) (Group_By lf)))
```

```
end.
```

The function `instance` associates to a relation name `r` its content. The function `sort` gives the sort of a query, that is to say the list of attribute names of the result of the query. In the case of the union, we first check if the sorts are the same for the sub-queries then if it is the case, we return the union of the bags (with possible duplicates) and if it is not, we return the empty bag. The function `Fbag.filter` allows to keep only the elements of a bag which satisfy some predicate. Here in the case of selection, our predicate evaluates a logical formula f on a tuple t thanks to the function `eval_q_formula`. The function `projection` projects each tuple of its sub-query (by mapping, with `Fbag.map`) on the `select_list` given as a first argument. The same idea applies to `Gamma`, not on tuples, but on groups of tuples computed by the function `make_groups`. Since it returns a list, it is transformed back into a bag by the function `Fbag.mk_bag`.

Since function, predicate and aggregate symbols are abstract at this stage, the semantics depends on axiomatized interpretation for them, respectively called `interp_symbol`, `interp_predicate` and `interp_aggregate`, that we will detail in the next section. We precise that predicates are interpreted in a three-valued logic, as standard in SQL with `NULL` values.

Throughout the evaluation, we need to keep track of groups of tuples, of the attributes associated to groups, and of the grouping expressions that are treated by the `projection` function, that will be used when evaluating sub-queries. This is the role of the environment `env`, which is very crucial to correctly handle correlated queries in SQL [3]. For a matter of clarity, we do not detail it here since it does not crucially impact our formalization. In addition to the `well_formed` predicate presented above to state that a query belongs to the fragment of aggregates we consider, [3] defines a predicate called `env_well_formed` stating that a query is well-defined in a given environment.

3.2 Instantiation

In this subsection, we present the main instantiation proposed in Datacert for the abstract syntax and semantics presented above.

Values can be either Booleans, integers and strings, with the `NULL` value represented by an `option` type. An `attribute` is represented by its name and its type. Given that, tuples are represented by a list of pairs of an attribute and a value.

The proof of concept also instantiates the function, predicate and aggregate symbols with the most standard of SQL: "+", "-" (unary and binary) and "*" for functions; "count", "avg", "sum" and "max" for aggregates; "=", "<=", … for predicates. The axiomatized functions `interp_symbol`, `interp_predicate` and `interp_aggregate` of the abstract layer are instantiated by interpreting the symbol "+" as an addition on integers, and so on.

4 FORMALIZATION: K-ALGEBRA AND ADEQUACY WITH RELATIONAL ALGEBRA

In this section, we present the first part of our contribution with the formalization in Coq of the semantics of a provenance-aware relational algebra, extended with top-level aggregates.

4.1 Semirings

To keep the spirit of Datacert, we also base the formalization of our mathematical structures on the Coccinelle [9] library. This library uses Coq `Record`s to represent objects of the structure and axioms on these objects.

This part of the formalization is rather standard, so we do not detail it here. We report on different possible implementations in the related works section (Sec. 6).

4.2 K-relations

As presented in **Definition 2**, the main idea is to replace the representation of the result of queries as bags of tuples by functions from tuples to a semiring K.

With only a function from tuples to K, we would lose crucial informations compared to bags. First, we would not know what is the sort of the relation. In the case of bags, we have access to the sort by picking a tuple in the bag (if it exists) and checking its sort, which is not the case for functions. Second, some operations of the algebra, such as the projection, iterate over the tuples whose annotation is non-zero. We thus need to know at least a superset of these tuples.

For these reasons, our definition of a K-relation is a `Record` with three fields: the function itself (named `f`), a `support` that encompasses at least tuples whose annotation is non-zero, and the sort of the relation `sort_krel`.

```
Record Krel : Type :=
  mk_kr
   {
     f : tuple ⟶ K;
     support : Fset.set tuple;
     sort_krel : Fset.set attributes;
   }.
```

In this definition, `Fset` is a library from Coccinelle of finite sets (similar to the `Fbag` library of the previous section).

To enforce the properties cited above, we define a `Record`, named `Krel_inv`, that states the invariant required for elements of type `Krel`. We will show later that, if this invariant is satisfied for queries consisting only of a relation, then it is enforced by all the operators of the algebra.

```
Record Krel_inv (kr:Krel) : Prop :=
  mk_kri
    {
     finite_support : ∀ t, t in? (support kr) = false ⟶ f kr t = 0;
     sort_labels_supp : ∀ t, t in? (support kr) = true ⟶ labels t = sort_krel kr;
    }
```

The field `finite_support` enforces that the support is actually a superset of the set of tuples that have a non-zero annotation. It is more general than having the exact set of tuples, and thus offers more possibilities for efficient implementations. The field `sort_labels_supp` ensures that the tuples of the support have the expected sort.

4.3 K-algebra

We now provide our formalization of the semantics of the K-algebra. As for the semantics of relational algebra presented in the previous section, it is a recursive function, called `eval_query_prov`. We also keep the two layers of abstraction of Datacert: the following function operates over abstract datatypes and objects, which will be refined in Sec. 5. In particular, as the semantics of the relational algebra is based on abstract interpretation functions for symbols, predicates and aggregates (respectively called `interp_symbol`, `interp_predicate` and `interp_aggregate`, see Sec. 3.1), our semantics is based on extensions of these functions to K-relations, respectively called `interp_symbol_prov`, `interp_predicate_prov` and `interp_aggregate_prov`. We will detail this latter function below.

```
Fixpoint eval_query_prov (env : env_prov) (q : query) : Krel :=
  match q with
  | Q_Table r ⟹ instance_prov r
| Q_Union q1 q2 ⇒
   let kr1 := eval_query_prov env q1 in
   let kr2 := eval_query_prov env q2 in
   if sort_krel kr1 =? sort_krel kr2
   then {| f := fun t ⇒ plus (f kr1 t) (f kr2 t);
           support := Fset.union (support kr1) (support kr2);
           sort_krel := sort_krel kr1 |}
   else Krel_empty
| Q_Sigma form q ⇒
let kr := eval_query_prov env q in
    {| f := fun t ⇒ mul (f kr t)
                    (if is_true (eval_formula_prov (env_pt env t) form) then 1 else 0);
       support := support kr;
       sort_krel := sort_krel kr |}
| Q_Pi s q ⇒
    let kr := eval_query_prov env q in
    {| f := fun t ⇒
         fold_left (fun x y ⇒ plus (f kr y) x)
           (filter (fun t' ⇒ eq? t (projection_prov (env_pt env t') s)) (Fset.elements (support kr))) 0;
       support := Fset.map (fun t ⇒ projection_prov (env_pt env t) s) (support kr);
       sort_krel := all_attr s |}
| Q_Empty_Tuple ⇒
  {| f := fun t ⇒ if t =? empty_tuple then 1 else 0;
     support := Feset.singleton empty_tuple;
     sort_krel := Fset.empty |}
| Q_NaturalJoin q1 q2 ⇒
   let kr1 := eval_query_prov env q1 in
   let kr2 := eval_query_prov env q2 in
   let sort := Fset.union (sort_krel kr1) (sort_krel kr2) in
   {| f := fun t ⇒ if labels t =? sort then mul (f kr1 t) (f kr2 t) else 0;
      support := natural_join_set (support kr1) (support kr2);
      sort_krel := sort |}
| Q_Gamma s lf f q ⇒ Krel_gamma s env (eval_query_prov env q) lf
end.
```

The function `instance_prov` has the same role as `instance_rel` except that it returns the associated K-relation instead of a bag of tuples. The `Q_Union` case directly matches the definition. For the selection, we keep the selected tuples by multiplying their annotations with 1, and 0 for the others; note that the support is not filtered and is thus a superset of the tuples. As we explained, for the projection, we have to iterate over the projections of non-zero tuples of the support, as in the mathematical definition. The function `all_attr` returns the attributes of the projected fields in s. The case of the empty tuple is straightforward. For natural join, we used an auxiliary function `natural_join_set`, that computes the natural join on sets of tuples (similar to `natural_join_bag` for bags). In this case, the resulting sort is the union set of the sort of the initial queries.

The more involved definition is for the aggregate operator `Q_Gamma`. Remember that `s` is the list of aggregates we actually compute (Sec. 3.1), and that we only handle the case were there is only one such aggregate (`well_formed` predicate). For other cases, since they will be ruled out by the `well_formed` hypothesis, we can return any K-relation.

```
Definition Krel_gamma s env kr lf :=
  match s with
  | agg::nil ⇒
    let lproj := make_proj_grb env lf (agg::nil) kr in
    let supp := Fset.mk_set (map fst lproj) in
    let sort := all_attr s in
    {| f := f_gamma lproj;
       support := supp;
       sort_krel := sort |}
  | _ ⇒ Krel_empty
  end.
```

We do not detail the definition of `make_proj_grb`, but it computes the projection of the support of the K-relation `kr` in a similar way as for the `Q_Pi` operator. The main difference is that it also groups the results by combining values and annotations as explained in **Definition 2**. Since this combination depends on aggregates, it is the role of the abstract `interp_aggregate_prov` function to perform this combination. This level of abstraction hides away the semimodule structure.

The other function appearing in this definition is the following:

```
Fixpoint f_gamma l :=
  match l with
  | nil ⇒ fun t ⇒ 0
  | (t0,kr) :: tl ⇒
    let f1 := f_gamma tl in
    fun t ⇒ if t0 =? t then (delta (map (f kr) (support kr))) + (f1 t) else f1 t
  end.
```

This function computes the annotation as presented in **Definition 2**, relying on a `delta` function of type:

```
delta : list K → K
```

Since this function depends on the semiring, which is abstract in this layer, we have to make this function abstract as well.

As for the semantics of the relational algebra presented in Sec. 3.1, we have an environment to correctly handle correlated queries, which is a simple extension of the environment of [3] to K-relations. We do not detail it in the article.

4.4 Adequacy Theorem

In this section we prove that our formalization of K-algebras faithfully extends the relational algebra: it exactly matches the $\mathbb{N}$-algebra. As presented in Section 2.4, other semirings for K thus give other provenance algebras.

Specializing to the semiring $\mathbb{N}$ fully determines the function `delta`: it returns `1` as soon as there is a non-zero annotation, and `0` otherwise [1].

```
Fixpoint delta (l : list N) : N :=
  match l with
  | nil ⇒ 0%N
  | hd :: tl ⇒ if hd =? 0%N then delta tl else 1%N
  end.
```

However, we took great care of keeping abstract all the other datatypes and functions that were axiomatized, in order to stay generic and ease the proof (see Sec. 4.5). It thus requires to assume some properties to relate them with their counterpart in the relational algebra. The first affected function is `instance_prov`: we need to know that the number of occurrences and the sorts coincide for each relation.

```
Hypothesis instance_prov_nb_occ : ∀ r t,
  (instance_prov r).(f) t = Fbag.nb_occ t (instance_rel r).
Hypothesis instance_support : ∀ r,
  sort_krel (instance_prov r) = basesort r.
```

In this definition, `Fbag.nb_occ` computes the number of occurrences of a tuple in a bag, and `basesort` returns the sort of a relation.

The second impacted functions are `interp_symbol_prov`, `interp_predicate_prov` and `interp_aggregate_prov`. We focus on the `interp_aggregate_prov` function. We recall that, in the relational algebra, the function

```
interp_aggregate : aggregate ⟶ list value ⟶ value
```

interprets the standard SQL aggregates by aggregating values into a single value. To generalize it to the K-algebra, it operates over annotated values: as presented in **Table 6** of the example of Sec. 1.1, the result of the query (and not only its annotation) depends on annotations of the initial tuples. Its type is thus:

```
interp_aggregate_prov : aggregate ⟶ list (value * K) ⟶ value
```

To relate both functions in the case of the semiring $\mathbb{N}$, we suppose that they coincide on lists in which all annotations are equal to one:

```
Hypothesis interp_aggregate_prov_prop1 : ∀ a l,
  List.∀ b (fun x ⟹ snd x =? 1%N) l = true ⟶
  interp_aggregate_prov a l = interp_aggregate a (fst (List.split l)).
```

where `List.split` splits a list of pairs into a pair of lists. Moreover, as `interp_aggregate` must be agnostic of the order of the elements in the list, a second property (that we omit here) of `interp_aggregate_prov` is that it must return the same result on two lists whose sum of annotations for each tuple is equal.

The last step is to make the two environments coincide. To do so, we can easily transform a list of tuples into a K-relation:

```
Fixpoint create_krel (l : list tuple) s :=
  match l with
  | nil ⟹ {| f := fun t ⟹ 0; support := Fset.empty ; sort_krel := s |}
  | hd :: tl ⟹
    let kr := create_krel tl s in
    {| f := fun x ⟹ if x =? hd then (f kr x) + 1 else f kr x;
       support := Fset.add hd (support kr);
       sort_krel := s |}
  end.
```

With all these hypotheses, we proved at the abstract layer that the two semantics coincide :

```
Theorem K_relations_extend_relational_algebra :
  ∀ env (b:bool) (q:query),
    well_formed b q = true ⟶
    env_well_formed env q = true ⟶
    ∀ (t:tuple),
      f (eval_query_prov_N (create_env env) q) t =
    Fbag.nb_occ t (eval_query_rel env q).
```

4.5 Discussion

While apparently straightforward once given, establishing this formalization was far from being trivial. One crucial aspect was to formulate simple but sufficient invariants on the K-algebra, as well as the properties on abstract datatypes presented above.

For instance, as shown in its type, the function `interp_aggregate` manipulates lists of values, and not bags. It implies that the same value can appear at multiple places in the list. On the K-relation side, we have functions, that immediately returns the number of occurrences. To smoothly relate the two, it was a long process by trial and error to end up with the type for `interp_aggregate_prov` that we just presented, as well as the two properties.

More generally, the mathematical definition of K-algebras involve non-trivial constructions, such as finite sums or the possibility to compute groups. We wanted to reflect these constructions in an executable way (see next section), which implies to manipulate actual data-structures, which sometimes obfuscates the proof.

On this aspect, we were very careful to preserve the high degree of abstraction inherited from Datacert. The direct impact is to have a generic formalization that can be instantiated after the fact in various ways. More interestingly, it results in a high-level formalization that is easier to carry out, since the various concerns are split into different parts. A striking example is that presentations of provenance for aggregates on paper insist a lot on the algebraic structure of semimodules required to compute results of aggregates [1]. Our degree of abstraction on values made it possible to hide this technicality during the abstract part of the formalization. The instantiation of `interp_aggregate_prov` actually provides the operations of the semimodules, but implicitly.

5 INSTANTIATION AND PROOF OF CONCEPT

5.1 Instantiation

The second part of our contribution is to provide instantiations of all the abstract objects that were axiomatized in the first part of the formalization. One of our main goal is to obtain an executable semantics, with reasonable performance after extraction. It allows one to compute provenance for SQL queries on small databases, and thus to actually test our implementation against other implementations, which could be really useful to detect bugs (which are usually unrelated to the size of the database).

First, we have implemented most of the usual semirings used in provenance presented in Sec. 2.4: Booleans and Boolean functions, natural and tropical integers, as well as polynomials. We insist in particular on two of these formalizations.

- Our formalization of tropical integers actually generalizes the definition of tropical integers to any totally ordered monoid. We have formally proved that the structure of a totally ordered monoid with compatibility of addition with respect to order is enough to be equipped with a tropical structure. That would allow us to reason in terms of linear cost to obtain data (tropical integers), but also to generalize it to more complex kinds of costs.
- Our formalization of polynomials refreshes the existing implementation of the Coccinelle [9] library. It is based on a new library of finite maps (which can be used in other contexts), to represent polynomials as maps from monomials to coefficients. This presentation of polynomials is very useful in our setting since it is both rather complete and executable.

Second, we have extended the Datacert proof of concept (Sec. 3.2) with our new functions, such as `interp_symbol_prov`, `interp_predicate_prov` and `interp_aggregate_prov`. It requires not only to implement them, but also to prove the properties that we have axiomatized in Sec. 4.4. In addition to providing a proof of concept, this realization thus validates the axiomatization.

5.2 Usability of the Implementation

To make this implementation easily usable, e.g. for quick testing, we implemented a Coq plugin that provides new Coq vernacular commands to parse SQL queries directly in Coq. For instance, the example of the introduction can be written as:

```
Parse_sql
  "create table table1 (name text, remission bool, treatment text, stage int);" t0.
Parse_sql
  "create table table2 (name text, remission bool, treatment text, stage int);" t1.
Parse_sql "insert into table1 values ...;" i0.
Parse_sql "insert into table2 values ..." i1.
Parse_sql "select count( * ) as c from (table table1 union table table2) table4
  where (stage ≤ 2 and remission = true) group by treatment;" q.
```

The first four commands create Coq terms `t0`, `t1`, `i0` and `i1` to populate the database (`...` must be replaced by the actual values). The last command creates the Coq term `q` containing the SQL `select` query.

Such examples can be found in the source code (see Sec. 1.2), in the file `data/proof_of_concept/Example.v`.

6 RELATED WORKS

One of the first formalizations of a data-oriented application is the implementation, in Agda, of a relational algebra, proposed in 2003 by [12, 13]. The first complete formalization of the relational model was proposed in [4], including a data model, the complete algebra, the semi-decision procedure *chase* and integrity constraints.

On the language side, the first verification of a relational database management system, in Coq, was presented in [16]. More complete SQL semantics were proposed in [2, 3]. We base on the second work which includes non trivial features of SQL such as set and bag semantics, null values, and correlated queries. A formalization of a query engine for SQL was presented in [5].

To our knowledge, this submission is the first work on data provenance formalized in a proof assistant. We base our work on a generalized framework for provenance for the relational algebra [14] and for aggregate functions [1]. The closest work [7] proposes a tool to decide the equivalence of two SQL queries based on a semantic called HottSQL, inspired by K-relations. However, their modeling exploits possibly infinite K-relations, which is not compatible with the SQL data model. Another limitation is that their semantics is not executable (not even after extraction, due to the infinite model), preventing from testing, comparing against other implementations, etc.

Provenance-aware query languages and database management systems are emerging, such as ProvSQL [18], an extension of PostgreSQL, or Links, a language-integrated query mechanism in Haskell [10, 19]. One of our mid-term objectives is to give strong guaranties to such systems by validating their trace analyses by a certified implementation.

Many formalizations of algebraic structures have been proposed in Coq, two of the most elaborate being Mathematical Components [20] and Math Classes[4]. Our work builds upon the Coccinelle library [9] for two reasons: it is based on [3]

[4]The Math Classes library can be found at https://github.com/coq-community/math-classes.

that already uses this library, and it is executable both in Coq and after extraction. Execution in Coq is useful for quick testing, in particular thanks to our plugin (see Sec. 5.2). To get more proof automation and to rely on a mainstream library, we consider switching to one of the two libraries listed above. For executability with the library Mathematical Components, one may use the refinement mechanism proposed in [8].

7 CONCLUSIONS AND PERSPECTIVES

We have presented a central basic block for the formalization of data provenance. Starting from existing formalizations of the extended relational algebra and algebraic mathematical structures, we have developed the provenance model based on K-relations for a non trivial algebra including aggregates, null values, and correlated queries. We have validated the implementation by proving the fundamental adequacy theorem with the standard relational algebra. Our Coq formalization offers a high level of abstraction, is parametric in the provenance semiring, but also comes with a proof of concept showcasing actual executions.

Short-term objectives include direct extensions of the language, such as nested aggregates. Pragmatically, this work can be linked with the formalization of SQL and its certified translation into the extended relational algebra [3] to propose a query language with annotations correctly and completely translated into the algebra. Another objective is to test our implementation against existing implementations of provenance such as the module ProvSQL of PostgreSQL [17].

Our main long-term perspective is to certify provenance in data analyses as performed in networks, bio-informatics, etc. Our case study are scientific workflows which, from input data, execute a sequence of instructions (with loops) to generate new data to be analyzed. We plan to instrument such workflows to produce execution traces to be checked in a certified tool to guaranty reproducibility [6, 11]: this *a posteriori* certification process will rely on a skeptical approach [15], and thus be independent from tools appearing in workflows, making it practicable and maintainable. The main challenge will consist in the scalability of the tool to be able to validate analyses on large amounts of data.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their time and for their comments.

REFERENCES

[1] Yael Amsterdamer, Daniel Deutch, and Val 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.
[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1555–1569, 2017.
[3] Véronique Benzaken and Évelyne Contejean. A coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 249–261. ACM, 2019.
[4] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A Coq Formalization of the Relational Data Model. In Zhong Shao, editor, *ESOP - 23rd European Symposium on Programming*, Lecture Notes in Computer Science, Grenoble, France, April 2014. Springer.
[5] Véronique Benzaken, Evelyne Contejean, Ch. Keller, and E. Martins. A coq formalisation of sql's execution engines. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 88–107. Springer, 2018.
[6] Sarah Cohen Boulakia, Khalid Belhajjame, Olivier Collin, Jérôme Chopard, Christine Froidevaux, Alban Gaignard, Konrad Hinsen, Pierre Larmande, Yvan Le Bras, Frédéric Lemoine, Fabien Mareuil, Hervé Ménager, Christophe Pradal, and Christophe Blanchet. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Comp. Syst.*, 75:284–298, 2017.
[7] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *PLDI*. ACM, 2017.

[8] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[9] Évelyne Contejean. Coccinelle, a Coq library for rewriting. In *Types*, Torino, Italy, March 2008.

[10] Stefan Fehrenbach and James Cheney. Language-integrated provenance by trace analysis. In Alvin Cheung and Kim Nguyen, editors, *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages, DBPL 2019, Phoenix, AZ, USA, June 23, 2019*, pages 74–84. ACM, 2019.

[11] Carole A. Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R. Crusoe, Kristian Peters, and Daniel Schober. FAIR computational workflows. *Data Intell.*, 2(1-2):108–121, 2020.

[12] C. Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *RelMiCS*, pages 137–148, 2003.

[13] C. Gonzalia. *Relations in Dependent Type Theory.* PhD thesis, Chalmers Göteborg University, 2006.

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

[15] J. Harrison and L. Théry. A Sceptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

[16] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[17] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql : Gestion de provenance et de probabilités dans postgresql. In *Proc. BDA*, Bucharest, Romania, November 2018. Conference without formal proceedings. (Demonstration).

[18] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. Provsql: Provenance and probability management in postgresql. In *Proc. VLDB*, pages 2034–2037, Rio de Janeiro, Brazil, August 2018. Demonstration.

[19] Jan Stolarek and James Cheney. Language-integrated provenance in haskell. *Art Sci. Eng. Program.*, 2(3):11, 2018.

[20] The Mathematical Components team. Mathematical Components. Available at https://math-comp.github.io/math-comp.