SQLCert: Coq mechanisation of SQL's compilation: Formally reconciling SQL and (relational) algebra

Véronique Benzaken, Evelyne Contejean

SQLCert: Coq mechanisation of SQL's compilation

Formally reconciling SQL and (relational) algebra

Véronique Benzaken and Évelyne Contejean

LRI - CNRS - Université Paris Sud - Université Paris Saclay, France

Abstract. SQL is *the* standard language for manipulating data stored in relational database systems. In theory, SQL is based on the *relational data model*. However, there is an important mismatch between the theoretical foundations and the corresponding standard specification, as SQL history spread over decades. Briefly, the disparities concern the treatment of relations: *finite sets* in theory, *finite bags* in practice, the treatment of attributes and the chosen corresponding algebra used to compile queries. We propose SQLCert, a Coq mechanisation of three, among four, central steps of SQL's compilation chain: the syntactic analysis, the semantics analysis and the logical optimisation steps. To this purpose, we propose SQL_{Coq} a Gallina grammar and associated Coq-mechanised semantics accounting for the native fragment of SQL described in the ISO/IEC 2006 Final Committee draft. As SQL compilers' logical optimisation is based on algebraic rewritings, we also define `ExtAlg` a Coq-mechanised extended bag-set-algebra, deeply relate SQL_{Coq} to it and prove, using Coq, most of the commonly used in practice (SQL's queries) rewritings, yielding strong guarantees for the optimiser. Doing so, we thus formally reconcile SQL and its theoretical algebraic counterpart and provide the *first*, to our knowledge, *executable mechanisation* proposal of a (*realistic fragment* of) SQL compiler.

1 Introduction

Current data-centric applications ranging from e-commerce, health crises' monitoring, to homeland security involve increasingly massive data volumes which are precious and whose availability, integrity and reliability is highly desirable. An important part of such data are handled by relational database management systems (RDBMS) through their query language: SQL which is *the* standard for such systems and whose ISO/IEC specification is found in [11]. RDBMS's, while intensively used in practice, have not yet reached the same high *safety* level guarantees as found in other critical systems, potentially yielding puzzling behaviours or even disastrous situations. Such a lack of strong assurance is problematic. Surprisingly, while formal methods are nowadays widely used to specify critical systems and to ensure that they comply with their specifications, such methods have not been broadly promoted for data-centric systems. Of course adopting such an approach in this context does involve taking into account a SQL compiler as an important piece in the chain and among formal methods, a promising way is to rely on the use of interactive theorem provers like Coq [18] or Isabelle [19]

More precisely, SQL compilation consists in four steps. The first two steps that include parsing and semantic analysis, translate the query in an algebraic expression. The last two steps also called the planning phase consist in logical and physical optimisation. The logical optimisation step exploits algebraic equivalences to perform sound query rewritings. The physical optimisation is in charge of producing query evaluation plans which are trees whose nodes are concrete, system-provided, implementations of algebraic operators. This last step is *data dependent* and is achieved based on auxiliary data structures and system maintained statistics.

According to textbooks [1], RDBMS and, thus, SQL are based, in theory, on the *relational data model.* However, there is an important mismatch between the theoretical foundations and the corresponding standard specification. Such a discrepancy is common but is even more serious in this context as SQL's history spreads over more than thirty years. Unlike what happened for "classical" programming languages, such as C let's say, in this particular context, the divergence has been *accentuated* due to the fact that SQL *not* being *Turing complete* more and more features have been added along the time (*e.g.*, aggregates[1]). Briefly, the disparities concern the treatment of relations: *finite sets* in theory, *finite bags* in practice. The treatment of attributes and the corresponding algebra used also diverge. In theory if attributes are *named* the corresponding algebra should be a *named set-algebra* and if only their positions are used an *unnamed set-algebra* is the correct corresponding one. In practice, attributes in SQL are *named* and have a *position* but the underlying algebra is an (almost) *unnamed bag-algebra.* Moreover, SQL syntax and semantics as described in the ISO/IEC JTC 1/SC 32 [11] document consist of thousands pages of informal specifications written in natural language. Obviously, it is hard to be convinced that the specification does have any theoretical algebraic counterpart and thus there are no *strong guarantees* that SQL compilers that do implement this specification do really comply with the theoretical foundations. To conclude, based on what is found in textbooks on the one hand and on the standardisation document on the other hand, any *usable in practice mechanisation* of SQL needs to:

1. faithfully handle a significant fragment of the language,
2. model relations and query results as *finite bags*,
3. carefully deal with *attributes names* and
4. provide and *rigorously relate* the considered fragment (*i.e.*, formally proving semantics's preservation) to an *extended algebra* that accounts for well-known query rewritings.

[1] an aggregate is an accumulator applied to a collection: `count, sum, avg, min, max`...

In addition, as it would not be realistic to handle SQL in its entirity[2], such a mechanisation should come together with a solid proof of concepts so as to convince users that it faithfully reflects the SQL's behaviours observed in mainstream systems.

Contributions In this article we propose SQLCert a formal framework that accounts for the three first compilation steps previously mentioned together with its proof of concept. SQLCert, handles the *native* fragment of SQL described page 315-398 in [11] *i.e.*, `select-from-where-group-by-having` statements with *function symbols, aggregates and nested queries*. To this end, we first define $\mathrm{SQL}_{\mathrm{Coq}}$ a SQL-friendly Gallina grammar that accounts for this fragment together with its associated *Coq mechanised semantics* $[\![\_]\!]_{\mathrm{SQL}_{\mathrm{coq}}}$. We also define a notion of well-formed $\mathrm{SQL}_{\mathrm{Coq}}$ queries. Each well-formed query being accepted by $[\![\_]\!]_{\mathrm{SQL}_{\mathrm{coq}}}$. Well-formedness forces queries to enjoy an algebraic counterpart. As such, it discards queries that are rejected by SQL, and also those that are unduly (lazily) accepted. This shall be made precise in Section 3.

Our second contribution consists in defining and formalising, using Coq, a *bag-set extended* algebra (`ExtAlg`) that is versatile enough to deal with function symbols with a predefined semantics (*e.g.*, SQL aggregates `avg,count,sum`) as well as user defined ones. By formally defining an *embedding* of (the named) relational algebra, mechanised in [4], into `ExtAlg` and rigorously proving its correctness, `ExtAlg` gracefully hosts the relational algebra. Unlike what is found in the literature, `ExtAlg` is very concise and *parametric* with respect to the data model and is the *first* to date mechanised *executable* bag-set algebra for SQL. We also proved, using Coq, most of (bag)-algebraic equivalences used in practice for logical optimisation, hence covering the third step of the compilation chain.

Our third contribution *formally* relates, using Coq, $[\![\_]\!]_{\mathrm{SQL}_{\mathrm{coq}}}$ to the semantics of `ExtAlg`, and proves the corresponding adequacy (semantics' preservation) theorem together with an Ocaml extraction, thus providing $\mathrm{SQL}_{\mathrm{Coq}}$ a *mechanised bag-set algebraic* counterpart. We also provide a proof of concept that allows to realise our abstract modelisation thus yielding an executable specification.

All those results yield a Coq mechanised compiler chain of $\mathrm{SQL}_{\mathrm{Coq}}$ and as such is an indispensable stage towards deeply specifying a SQL compiler. Such a compiler chain is the *first*, to our knowledge, *executable mechanisation* proposal of a (*realistic fragment* of) SQL compiler able to *formally* reconcile SQL with its *algebraic theoretic foundations*.

Organisation In Section 2 we first remind relational algebra. Section 3 briefly presents SQL, relating it with its algebraic counterpart, and precisely detail, through examples, the discrepancy between the theoretical foundations and the specification, evidencing surprising behaviours encountered. $\mathrm{SQL}_{\mathrm{Coq}}$ is detailed in Section 4. Section 5 presents our extended algebra, the embedding, its correctness proof as well as the soundness proof of many rewritings used in practice.

[2] The standardisation document only concerning SQL is more than 1300 pages long!

Section 6 details our SQL's compiler mechanisation together with its proof of adequation. We compare our contributions with related works, conclude, drawing lessons, and give perspectives in Section 7.

2 Theory: the relational model

The relational model serves different related purposes: it allows to *represent* information through *relations* and to *refine* the represented information by further restricting it through *integrity constraints*. It also provides ways to *extract* information through *query languages* based on algebra[3]. Relational algebra consists of a set of operators with relations as operands. We briefly recall the basics as found in [1]. Intuitively, in the relational model, data is represented by tables (*relations*) consisting of rows (*tuples*), with uniform structure and intended meaning, each of which gives information about a specific entity. Tuples have a *support* which is a finite set of fields together with their corresponding basic type. The columns of a table also have names, called *attributes*. Each attribute is associated with its corresponding *domain* (noted *dom()*) which is a basic (flat) type. In practice, the structure of a table is given by a relation *name* and a *finite set* of attributes: its *sort*. Its contents, *i.e.*, the *finite* set of tuples populating it, is referred to as the *instance* of the relation. For a tuple t to belong to relation r the *well-sortedness* condition $\mathit{support}(t) = \mathit{sort}(r)$ must hold. Two different equivalent versions of the model exist: the *unnamed* and the *named* ones. Whether we place ourselves in a named or unnamed perspective different algebraic operators are considered.

2.1 Unnamed setting

In the *unnamed* setting, the specific attributes of a relation are ignored: only their position is available to query languages. Three primitive algebraic operators form the unnamed algebra: selection, projection and Cartesian product. This algebra is more often referred to as the *SPC algebra*.

$$q := r \mid \sigma_f(q) \mid \pi_W(q) \mid q \times q$$

We define the operators forming the SPC algebra. First, base relations, r, are queries. The two primitive forms for the selection condition f over an expression q of arity n are $j = c$ and $j = k$, where j, k are positive integers $\leq n$ and $c \in \mathit{dom}(j)$. The semantics is given by $[\![\sigma_{j=c}(q)]\!] = \{t \mid t \in [\![q]\!] \wedge t_j = [\![c]\!]\}$ where t_j is the j^{th} component of tuple t. The operator $\sigma_{j=k}$ is defined analogously. Projection π can be used to delete and/or permute columns of an expression. The general form of this operator is π_W , where W is a possibly empty sequence, $j_1, \ldots, j_n$ of positive integers, possibly with repeats. This operator takes as input any expression with arity $\geq \mathit{max}(j_1, \ldots, j_n)$ (where the max of $\emptyset$ is 0) and returns an expression with arity n whose semantics is $[\![\pi_{j_1,\ldots,j_n}(q)]\!] = \{(t_{j_1}, \ldots, t_{j_n}) \mid t \in [\![q]\!]\}$. The Cartesian product provides the capability for combining expressions.

[3] or first-order logic.

It takes as inputs a pair of expressions having arbitrary arities n and m and returns an expression with arity $n+m$. $[\![q_1 \times q_2]\!] = \{(t_1, \dots, t_n, s_1, \dots, s_m) \mid t \in [\![q_1]\!] \wedge s \in [\![q_2]\!]\}$. Cross-product is associative and non-commutative and has the non empty 0-ary relation $\{()\}$ as left and right identity.

2.2 Named setting

In the *named* setting, attributes are viewed as an *explicit part* of the database. They are used by the query language. Obviously, for modelling purposes, names carry much more information than column numbers, this explains why relational systems use attributes' names rather that positions. Four operators form the SPJR algebra:

$$q := r \mid \sigma_f(q) \mid \pi_W(q) \mid \rho_g(q) \mid q \bowtie q$$

Again, in this setting, base relations, r are expressions. Concerning the selection operator, in textbooks, it has the form $\sigma_{\mathtt{a}=c}$ or $\sigma_{\mathtt{a}=\mathtt{b}}$, where $\mathtt{a}, \mathtt{b} \in$ *attribute* and $c \in$ *dom(*$\mathtt{a}$*)*. The notation $\mathtt{a} = c$ ($\mathtt{a} = \mathtt{b}$ resp.,) is improper and corresponds to $x.\mathtt{a} = c$ ($x.\mathtt{a} = x.\mathtt{b}$ resp.,) where x is a free variable. The selection applies to any expression q of sort S, (with $\mathtt{a}, \mathtt{b} \in S$) and yields an expression of sort S. The semantics of the operator is $[\![\sigma_f(q)]\!] = \{t \mid t \in [\![q]\!] \wedge [\![f]\!]\{x \to t\}\}$ where $[\![f]\!]\{x \to t\}$ stands for "t satisfies formula $[\![f]\!]$", x being the only free variable of $[\![\mathrm{f}]\!]$. Formula satisfaction is based on the standard underlying interpretation.

The projection operator has the form $\pi_{\{\mathtt{a}_1,\dots,\mathtt{a}_n\}}$, $n \geq 0$ and operates on all expressions, q, whose sort contains the subset of attributes $W = \{\mathtt{a}_1, \dots, \mathtt{a}_n\}$ and produces an expression of sort W. The semantics of projection is $[\![\pi_W(q)]\!] = \{t|_W \mid t \in [\![q]\!]\}$ where the notation $t|_W$ represents the tuple obtained from t by keeping only the attributes in W. The natural join operator, denoted $\bowtie$, takes arbitrary expressions q_1 and q_2 having sorts V and W, respectively, and produces an expression with sort equal to $V \cup W$. The semantics is, $[\![q_1 \bowtie q_2]\!] = \{t \mid \exists v \in [\![q_1]\!], \exists w \in [\![q_2]\!], t|_V = v \wedge t|_W = w\}$. It is *important to notice* that when $\mathit{sort}(q_1) = \mathit{sort}(q_2)$, then $[\![q_1 \bowtie q_2]\!] = [\![q_1]\!] \cap [\![q_2]\!]$, and when $\mathit{sort}(q_1) \cap \mathit{sort}(q_2) = \emptyset$, then $[\![q_1 \bowtie q_2]\!]$ is the cross-product of $[\![q_1]\!]$ and $[\![q_2]\!]$ ($[\![q_1]\!] \times [\![q_2]\!]$). The join operator is associative and commutative[4]. An attribute renaming for a finite set V of attributes is a one-one mapping from V to *attribute*. In textbooks, an attribute renaming g for V is specified by the set of pairs $(\mathtt{a}, g(\mathtt{a}))$, where $g(\mathtt{a}) \neq \mathtt{a}$; this is usually written as $\mathtt{a}_1\mathtt{a}_2 \dots \mathtt{a}_n \to \mathtt{b}_1\mathtt{b}_2 \dots \mathtt{b}_n$ to indicate that $g(\mathtt{a}_i) = \mathtt{b}_i$ for each $i \in [1, n], n \geq 0$. A renaming operator for expressions over V is an expression ρ_g, where g is an attribute renaming for V; this maps to outputs over $g[V]$. Precisely, for q over V, $[\![\rho_g(q)]\!] = \{v \mid \exists u \in [\![q]\!], \forall \mathtt{a} \in V, v(g(\mathtt{a})) = u(\mathtt{a})\}$.

[4] This seems to contradict the fact that cross product is not commutative in the SPC setting. Notice that in such a setting, if $t = (1, 2)$ and $t' = (3, 4)$, $\{t\} \times \{t'\}$ and $\{t'\} \times \{t\}$ are different, whereas in an SPJR setting, if $t(\mathtt{a}_1) = 1, t(\mathtt{a}_2) = 2$ and $t'(\mathtt{a}_3) = 3, t'(\mathtt{a}_4) = 4$, their combination, whatever the order, is the function tt' defined by $tt'(\mathtt{a}_1) = 1, tt'(\mathtt{a}_2) = 2, tt'(\mathtt{a}_3) = 3, tt'(\mathtt{a}_4) = 4$.

2.3 Adding union, intersection and difference

Though union, intersection and difference are not part of the SPC and SPJR minimal algebras, we include them as they are part of SQL. As standard in mathematics, $q_1 \cup q_2$ (resp. $q_1 \cap q_2, q_1 \setminus q_2$) is the set containing the union (resp., intersection, difference) of the two sets of tuples. The subtle point is that these set operators can only be applied over sets of tuples with the *same sort.*

3 Reality: SQL

There is an important mismatch between the theoretical foundations and the corresponding standard specification. Such a gap concerns *(i)* the structure of attributes, tuples, relations and query sorts, *(ii)* the nature of relations' contents and *(iii)* the relationship between queries and algebra. It has puzzling impacts, as will be made explicit in Section 3.3 and 3.4, that any faithful and accurate mechanisation has to account for.

3.1 Attributes, tuples and relations: named and unnamed settings

Quoting page 51 of the ISO document attributes are specified by:

> "*The terms column, field, and attribute refer to structural components of tables, row types, and structured types, [...] in analogous fashion. As the structure of a table consists of one or more columns, so does the structure of a row type consist of one or more fields [...] Every structural element, whether a column, a field, or an attribute, is primarily a* **name** *paired with a declared type. The elements of a structure are* **ordered**. *Elements in different* **positions** *in the same structure can have the same declared type but* **not the same name**. *[...] in* **some circumstances** *[...] the compatibility [...] is determined solely by considering the declared types of each pair of elements at the same* **ordinal position**."

Fig. 1. ISO: attributes and tuples

The specification makes a difference between attributes of a relation called "columns", attributes of a tuple called "fields" and attributes of structured (user-defined) types which are called in turn "attribute". In any case, attributes (columns or fields or attributes in the specification) are named *and* have an ordinal position and relations' sorts in SQL are ordered lists with no duplicates and according to p 322 in the document, queries' sorts are ordered lists allowing for attributes' names duplication[5], both in sharp contrast with the relational model. Under *some circumstances* covers queries that involve a set operator such as `union, intersect, except` for which attributes names are simply forgotten. Further, SQL's table's contents, called collections in page 53 of the ISO document, allow for element duplication (page 56) in contrast with *finite sets.* At that point, it clearly appears that SQL enjoys both attributes' names and positions

[5] "Let C be some column. Let TE be the <*table expression* >. C is an underlying column of TE if and only if C is an underlying column of some column reference contained in TE. "

and does not consider instances of relations as finite sets but rather collections allowing for duplicates.

3.2 SQL queries: SPJR and SPC

A classical SQL query consists of a `select-from-where` block that can be extended with a `group-by-having` clause. A SQL query returns a collection and this is why the language is often considered to enjoy a bag (or multiset) semantics[6]. The `distinct` keyword is used to force it to *mimic* a set semantics while the keyword `all` to force a bag semantics. However, the term "semantics" in this particular context is improperly used. It rather covers query's membership. At that point, *as long as aggregates, functions and difference*[7] operators are *not used*, SQL is not duplicate sensitive. More precisely, if one add or remove the keyword `distinct` or `all` for all SQL operators in a query q , yielding $q_{\texttt{distinct}}$ and $q_{\texttt{all}}$ this does not affect the membership relation (*i.e.*, the fact that a tuple appears at least once in the result) for query evaluation:

$$\forall t, t \in [\![q]\!]_{\text{SQL}} \iff t \in [\![q_{\texttt{distinct}}]\!]_{\text{SQL}} \iff t \in [\![q_{\texttt{all}}]\!]_{\text{SQL}}$$

Of course, the tuple's multiplicity is affected. As previously stated, there is a tight link between SQL and its algebraic counterpart as illustrated through examples in Figure 2. We assume the following database schema which contains relations `tbl0(a,b,c)` `tbl1(a,b,c)` and `tbl2(d,e,f)`. We further assume that all attributes vary in a unique domain: `int`. The first two queries return all the tuples in relation `tbl0`, for which the `where` clause `b>3` is satisfied. Indeed they respectively correspond to the algebraic expressions $\pi_{\{\texttt{a,c}\}}(\sigma_{\texttt{b}>3}(\texttt{tbl0}))$. and $\rho_{\{\texttt{a}\to\texttt{a1};\texttt{c}\to\texttt{c1}\}}(\pi_{\{\texttt{a,c}\}}(\sigma_{\texttt{b}>3}(\texttt{tbl0})))$. The third query details how relations can be combined through the `from` part of `select- from-where` blocks. However, rather than computing a join ($\bowtie$), as would be expected in a named setting, a cross product is used instead and the resulting query's sort issued by the system

(1) `select a, c from tbl0 where b>3;`	$\pi_{\{\texttt{a,c}\}}(\sigma_{\texttt{b}>3}(\texttt{tbl0}))$
(2) `select a as a1, c as c1` `from tbl0 where b>3;`	$\pi_{\texttt{a1,c1}}(\rho_{\{\texttt{a}\to\texttt{a1};\texttt{c}\to\texttt{c1}\}}(\sigma_{\texttt{b}>3}(\texttt{tbl0})))$
(3) `select * from tbl0,tbl1;`	$\texttt{tbl0}\times\texttt{tbl1}$
(4) `select * from tbl1, (select d, f from tbl2) as t2` `where b=d;`	$\texttt{let t2} = \pi_{\{\texttt{d,f}\}}(\texttt{tbl2})$ $\texttt{in } \sigma_{\texttt{b=d}}(\texttt{tbl1} \times \texttt{t2})$
(5) `select * from tbl1` `where tbl1.c in (select tbl2.e from tbl2);`	$\pi_{\texttt{a,b,c}}(\sigma_{\texttt{c=e}}(\texttt{tbl1}\times\texttt{tbl2}))$

Fig. 2. ISO SQL's queries and their algebraic counterpart

[6] Even if tuples's multiplicities are not part of the tuple definition nor are they primitive in SQL.

[7] Such is the case for the `select-from-where-group-by-having`

is {`a,b,c,a,b,c`}. The algebraic counterpart: `tbl0` × `tbl1` is thus improper as it induces attribute's name collision. According to SQL's `from` clause specification (p323-324 of [11]), shown in Figure 3, it seems that a cross product is indeed used. Obviously, it is hard to be convinced that the specification corresponds to a cross product and thus there are no strong guarantees that SQL compilers, that do implement this specification, really comply with its algebraic counterpart. The next query in Figure 2 illustrates the fact that queries can be combined through bindings to fresh names yielding algebraic expressions up to β-reduction. The last example illustrates the fact that SQL's `where` conditions and relational algebra's formulae do not match though an algebraic expression can still be assigned to such queries.

1. *Let TRLR be the result of TRL Case:*
   - *(a) If TRL simply contains a single* `<table reference>` *TR then TRLR is the result of TR.*
   - *(b) If TRL simply contains n* `<table reference>` *s, where $n > 1$, then let TRL-P be the* `<table reference list>` *formed by taking the first n1 elements of TRL in order, let TRL-L be the last element of TRL, and let TRLR-P be the result of TRL-P. If TRLR-P contains m rows, $m \geq 1$ (one), then for every row Ri 1 (one)$\leq$ i $\leq$ m let TRLR-Li be the corresponding evaluation of TRL-L under all outer references contained in TRL-L Let SUBRi be the table containing every row formed by concatenating Ri with some row of TRLR-Li Every row RR in SUBRi is a row in TRLR, and the number of occurrences of RR in TRLR is the sum of the numbers of occurrences of RR in every occurrence of SUBRi . The result of the* `<table reference list>` *is TRLR with the columns reordered according to the ordering of the descriptors of the columns of the* `<table reference list>` *.*
2. *The result of the* `<from clause>` *is TRLR*

Fig. 3. ISO: SQL's `from` clause specification

```
(6) select * from tbl1 where (c >= all (select b from tbl1));
(7) select avg(a+c) from tbl1;
(8) select 2*(a+c), sum(a) from tbl1 group by a+c, b having b > 6;
```

Fig. 4. ISO SQL's queries with no algebraic counterpart

Queries on Figure 4 do not fall in the relational algebra fragment because they use either function symbols in the `select` (or `where`) clause (`avg(a+c)`) or in a `group-by-having` clause[8], quantifiers in the `where` clause (`c >= all`) thus having no relational algebra counterpart. Membership characterisation for the first query corresponds to:

$$\{x \mid x \in \texttt{tbl1} \; ; \; \forall \, y \in \texttt{tbl1}, \; \texttt{c}(x) \geq \texttt{b}(y)\}$$

which has no algebraic counterpart. The second query computes the average value of the mono-column table resulting of computing, for each tuple occurring in `tbl1`, the average of the sum of attributes `a` and `c`. Last, let's grasp the behaviour of the `group-by-having` clause as it is specified page 345 of the ISO

[8] `group-by-having` are used with aggregates if not they correspond to a `select` block.

document. In a first step, the `group-by` clause minimally partitions `tbl1` (or more generally the relation resulting of the evaluation of the `from` clause) into several homogeneous groups according to the values of the expressions $e_1, \ldots, e_n$ following the `group-by` (`a+c` and `b`). Homogeneous means that all tuples in a given group have the same values for the e_i's. In a second step, each group yields a single tuple, computed via the expressions e'_k's occurring in the `select` part (`2*(a+c)` and `sum(a)`). For the whole query being accepted, these expressions have to be built only upon functions applied over e_i's or aggregates applied without any restrictions[9]. Then groups (and hence a final tuple) can be discarded by the `having` clause, which is a logical formula built upon expressions with the same restrictions as for the e'_k's.

3.3 Sorts' mismatch and attributes' ambiguity

The following queries illustrate puzzling behaviours related to the fact that sorts are not handled as sets. More precisely the first `select` allows a collapsing "renaming" from both `a` and `b` to `c` which produces a result which is not, in theory, a relation since its sort is not a set

(1)	`select a as c,b as c from tbl1;`	(2)	`select * from tbl0, tbl1;`

(3)	`select a from tbl0, tbl1;`
	`ERROR: column reference "a" is ambiguous`
		(4)	`select tbl0.a from tbl0, tbl1;`

(5)	`select a from`
	`(select * from tbl0, tbl1) tbl;`
	`ERROR: column reference "a" is ambiguous`
	`LINE 1: select a from`
	`(select * from tbl0, tbl1) tbl;`
		(6)	`select tbl0.a from`
			`(select * from tbl0, tbl1) tbl;`
			`ERROR: missing FROM-clause entry for`
			`table "tbl0"  LINE 1: select tbl0.a`
			`from (select * from tbl0, tbl1) tbl;`

(7)	`select tbl0a  from(select * from tbl0, tbl1) t3(tbl0a,tbl0b,tbl0c,tbl1a,tbl1b,tbl1c);`		

Fig. 5. Sorts' mismatch and attributes' ambiguity

As previously stated the evaluation of the second query is not a join but a Cartesian product but its resulting sort is {`a`,`b`,`c`,`a`,`b`,`c`}. This is admitted because internally attribute's names are prefixed by the relation's name they are attached to, hence being pairwise distinct. When there is no mean to distinguish between two columns with the same name, the system cannot assign a semantics to `from` and complains. This is illustrated by query (3) which is rejected and its reformulation (4) which is accepted.

However, queries as the second one, accepted at top level, while they should be, in theory, discarded, are adequately rejected as sub-queries only on a by-need basis, as illustrated by (5). One could expect that the same solution than the

[9] This restriction ensures that a group will provide a single *flat* tuple. Whenever another expression should occur in the `select` part, let's say `a`, a group may contain several distinct values for it, and would provide a set of values for `a`.

one taken previously for disambiguating the query in this context should work. Unfortunately such is not the case (as shown by (6)). Since SQL is unable to correctly manage an environment it is impossible to precisely point an attribute when it comes from an inner query as mentioned page 329 of the ISO document. The only way to get the query accepted is to reformulate it explicitly renaming attributes as expressed by query (7).

3.4 Non linearity

Let us further comment about the actual semantics of the `from` clause and consider the query in Figure 6, a non linear variant of the second query in Figure 5. The remark we made about internal disambiguation of attributes explains why it is impossible to build an auto Cartesian product. But this is baffling as SQL actually detects this case but refuses to assign it a corresponding algebraic expression.

```
select * from tbl0, tbl0;
ERROR: table name "tbl0" specified more than once
```

Fig. 6. Non linearity

However, the situation is more subtle than a mere scoping problem. Basically, it is closely related to bags and to the fact that SQL mixes SPJR and SPC algebras. Indeed, there are two potential semantics for this query whether one wants to preserve tuple's multiplicities or only membership. In other words, in theory, in a set theoretic setting, auto join and intersection coincide but such is not the case for bags for which this property does not hold: multiplicities are multiplied for cross products and joins while for intersection they correspond to a min. SQL's does not want to favour one algebra w.r.t., another and thus rejects the query.

3.5 Other SQL features: null values, outer joins, order-by

SQL provides `NULL` values. Such values could seem tricky to handle but they will be dealt with by simply considering them as absorbing elements in expressions and by defining a three-valued logic for formulae. For this reason we shall not deal for the moment with `outer joins` as they make intensive use of `NULL` values. Last we do not handle `order by` clause nor ranking aspects (`limit` for instance). While used in practice, they are not central to this work. Moreover, they require collections to be equipped with an order. All those features will be taken into account in future work.

3.6 Assessment

SQL relations and result of queries are not finite sets but finite collections allowing for duplicates. Such an option was historically made for performance reasons as duplicate elimination is an expensive task. This early choice was not harmful as long as duplicate sensitive constructs were not present. But as the language evolved over years, including more and more features, among those *aggregates*, it happened that SQL queries results were *duplicate sensitive* and particular attention has to be dedicated to handle this situation cleanly and faithfully (especially in the context of query rewriting).

As we illustrated, SQL provides attributes as denotable entities. This suggests that a named SPJR version of the algebra should underly the language's semantics. Unfortunately, according to the ISO specification, SQL underlying algebra seems to be, with no strong guarantees, SPC though the only way to denote columns is through attributes' names. This introduces another foundational mismatch in the language yielding potential bugs as it is under application programmers' responsibility to manage names in an unnamed setting. Again, we insist, any decent, accurate, mechanisation of SQL has to manage attributes very carefully.

4 SQLCert

We now present the SQLCert framework. SQLCert handles SQL's collections as bags and provides $\mathrm{SQL}_{\mathrm{Coq}}$ a name-based SQL-compliant Gallina grammar together with its Coq formalised semantics that will be, in Section 6, *formally connected* to a *bag-set algebra*. In particular, $\mathrm{SQL}_{\mathrm{Coq}}$ sticks to the ISO standard and, thus, faithfully reflects the aforementioned SQL's puzzling situations.

4.1 $\mathrm{SQL}_{\mathrm{Coq}}$: syntax

$\mathrm{SQL}_{\mathrm{Coq}}$ is written in Gallina and takes into account nested SQL queries with aggregates and function symbols and assigns them a Coq mechanised semantics. For the sake of clarity, we choose to present it as an abstract syntax. More precisely, $\mathrm{SQL}_{\mathrm{Coq}}$ grammar is given by (where α denotes an attribute):

$$
\begin{array}{lll}
sq & ::= & \texttt{table}\ name \\
 & & \mid \texttt{select}\ (* \mid \overrightarrow{e^a\ \texttt{as}\ \alpha})\ \texttt{from}\ \overrightarrow{sq[r]}\ \texttt{where}\ F\ \texttt{group by}\ (\texttt{singleton} \mid \overrightarrow{e^f})\ \texttt{having}\ F \\
 & & \mid sq\ \texttt{union}\ sq \ \mid\ sq\ \texttt{intersect}\ sq \ \mid\ sq\ \texttt{except}\ sq \\
r & ::= & * \ \mid\ \overrightarrow{\alpha\ \texttt{as}\ \alpha} \\
f & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{sqrt} \mid \texttt{sin} \mid \ldots \mid \textit{user defined function} \\
a & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Avg} \mid \textit{user defined aggregate} \\
e^f & ::= & value \mid \alpha \mid f(\overrightarrow{e^f}) \\
e^a & ::= & e^f \mid a(e^f) \mid f(\overrightarrow{e^a}) \\
F & ::= & F\ \texttt{and}\ F \mid F\ \texttt{or}\ F \mid \texttt{not}\ F \mid A \\
A & ::= & \texttt{true} \mid p(\overrightarrow{e^a}) \mid p(\overrightarrow{e^a}, \texttt{all}\ sq) \mid p(\overrightarrow{e^a}, \texttt{any}\ sq) \mid (* \mid \overrightarrow{e^a\ \texttt{as}\ \alpha})\ \texttt{in}\ sq \\
p & ::= & \texttt{=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \mid \textit{user defined predicate}
\end{array}
$$

We tried, as far as possible to stick to SQL's syntax but the SQL-aware reader shall notice that $\mathrm{SQL}_{\mathrm{Coq}}$ differs from SQL in different ways. First, for the sake of uniformity, we impose to have the whole `select-from-where-group-by-having` construct (no optional `where` and `group-by-having` clauses). When the `where` clause is empty, it is forced to `true`. Similarly, as the `group-by` clause partitions the collection of tuples obtained evaluating the `from` clause, when no `group-by` is present in SQL, we force $\mathrm{SQL}_{\mathrm{Coq}}$ to work with the finest partition[10] which corresponds to the `singleton` case. We also force explicit and mandatory renaming of attributes, when $*$ is not used. In our syntax, `select a, b from tbl1;` is expressed by `select a as a, b as b from (table tbl1[∗]) where true group-by singleton having true`. A further, more subtle, point worth to mention is the distinction we make between e^f and e^a. Both are expressions but the former are built only with function symbols (f) and are evaluated on *tuples* while the latter also allow unested[11] aggregates symbols (a) and are, in that case, evaluated on *collections of tuples*. Only e^f are used by the `group-by` so as to generate uniform groups (as it is the case in SQL). In the same line, we used the same language F for formulae either occurring in the `where` (dealing with a single tuple) or in the `having` clause (dealing with collections of tuples) simply by identifying each tuple with its corresponding singleton. Also, no aliases for queries are allowed.

```
 select ∗  from tbl1 as t1(a1,b1,c1), tbl1 as t2(a2,b2,c2) where a1 = a2;
```

is expressed by:

```
select ∗  from (table tbl1[a as a1, b as b1, c as c1],
               table tbl1[a as a2, b as b2, c as c2])
where a1 = a2 group by singleton having true
```

Indeed, when attributes are properly renamed, query aliases become useless, hence we choose to not use them in our syntax. This syntax captures admissible SQL queries such as:

```
select * from tbl1
where a+b >= all (select (tbl0.a+tbl1.c) from tbl0, tbl1);

select a, count(b) from tbl1 group by a
having avg(c) >= all (select a from tbl1) ;
```

which are expressed (omitting the `group by singleton having true`) by:

```
select * from tbl1[*]
where a+b >=  all (select (a0 + c1) as a0_plus_c1
                  from tbl0[a as a0, b as b0, c as c0],
                       tb11[a as a1, b as b1, c as c1]);


select a as a, count(b) as countb from tbl1[*] group by a
having avg(c) >= all (select a as a from tbl1[*]);
```

[10] The partition consisting of the collection of singletons, one singleton for each tuple in the result of the `from`

[11] e^a is of the form: `avg(a)`; `sum(a+b)`; `sum(a+b)+3`; `sum(a+b)+avg(c+3)` but not of `avg(sum(c)+a)`

This mentioned, $\mathrm{SQL}_{\mathrm{Coq}}$ matches SQL. In particular, at that point, attribute ambiguities are still possible. In order to avoid the related problems mentioned in Section 3 and to accurately account for SQL, while being compliant with an algebraic model, we shall introduce, in Figure 7, the definition of *well-formed* ($\mathrm{SQL}_{\mathrm{Coq}}$) queries which relies, in turn, on the notion of query *sort*. Each well-formed $\mathrm{SQL}_{\mathrm{Coq}}$ query will enjoy an algebraic counterpart.

$$
\begin{array}{l}
\mathtt{WF}(\mathtt{table}\ \ n) = \mathit{true} \\
\mathtt{WF}(sq_1 \Box sq_2) = \mathtt{WF}(sq_1) \wedge \mathtt{WF}(sq_2) \wedge \mathit{sort}(sq_1) = \mathit{sort}(sq_2) \\[1ex]
\mathtt{WF}(\mathtt{select}\ \overrightarrow{e^a\ \mathtt{as}\ \alpha}\ \mathtt{from}\ \overrightarrow{sq_j[r_j]}\ \ \mathtt{where}\ F_1\ \mathtt{group\ by\ singleton\ having}\ F_2) = \\
\qquad \mathtt{WF}(\overrightarrow{sq_j[r_j]}) \wedge \mathtt{WF}_s(F_1) \wedge \mathtt{WF}_s(F_2) \wedge \mathit{pairwise}_{\neq}(\overrightarrow{\alpha}) \wedge \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \\
\qquad\qquad\qquad \text{if } s = \bigcup_j \mathit{sort}(sq_j[r_j]) \\[1ex]
\mathtt{WF}(\mathtt{select}\ \overrightarrow{e^a\ \mathtt{as}\ \alpha}\ \mathtt{from}\ \overrightarrow{sq_j[r_j]}\ \ \mathtt{where}\ F_1\ \mathtt{group\ by}\ \overrightarrow{e^f}\ \mathtt{having}\ F_2) = \\
\qquad \mathtt{WF}(\overrightarrow{sq_j[r_j]}) \wedge \mathtt{WF}_s(F_1) \wedge \mathtt{WF}_s(F_2) \wedge \mathit{pairwise}_{\neq}(\overrightarrow{\alpha}) \wedge \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \wedge \\
\qquad \mathcal{A}(\overrightarrow{e^f}) \subseteq s \wedge \bigwedge_{e^a} \mathit{builtupon}(e^a, \overrightarrow{e^f}) \wedge \mathit{builtupon}(F_2, \overrightarrow{e^f}) \\
\qquad\qquad\qquad \text{if } s = \bigcup_j \mathit{sort}(sq_j[r_j]) \\[1ex]
\mathtt{WF}(\mathtt{select}\ \ *\ \ \mathtt{from}\ \overrightarrow{sq_j[r_j]}\ \ \mathtt{where}\ F_1\ \mathtt{group\ by}\ G\ \mathtt{having}\ F_2) = \\
\qquad \mathtt{WF}(\mathtt{select}\ \overrightarrow{(a_i\ \mathtt{as}\ a_i)}_{a_i \in s}\ \mathtt{from}\ \overrightarrow{sq_j[r_j]}\ \ \mathtt{where}\ F_1\ \mathtt{group\ by}\ G\ \mathtt{having}\ F_2) \\
\qquad\qquad\qquad \text{if } s = \bigcup_j \mathit{sort}(sq_j[r_j]) \\[1ex]
\mathtt{WF}(\overrightarrow{sq_j[r_j]}) = \bigwedge_j \mathtt{WF}(sq_j[r_j]) \wedge \mathit{pairwise}_{\cap=\emptyset}(\overrightarrow{\mathit{sort}(sq_j[r_j])}) \wedge \mathit{pairwise}_{\neq}(\overrightarrow{sq_j[r_j]}) \\
\mathtt{WF}(sq[*]) = \mathtt{WF}(sq) \\
\mathtt{WF}(sq[\overrightarrow{b_i\ \mathtt{as}\ a_i}]) = \mathit{pairwise}_{\neq}(\overrightarrow{a_i}) \wedge \bigcup_i \{b_i\} = \mathit{sort}(sq) \wedge \mathtt{WF}(sq)
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{WF}_s(F_1\ \ \mathtt{and}\ \ F_2) = \mathtt{WF}_s(F_1) \wedge \mathtt{WF}_s(F_2) & \mathtt{WF}_s(p(\overrightarrow{e^a})) = \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \\
\mathtt{WF}_s(F_1\ \ \mathtt{or}\ \ F_2) = \mathtt{WF}_s(F_1) \wedge \mathtt{WF}_s(F_2) & \mathtt{WF}_s(p(\overrightarrow{e^a},\ \mathtt{all}\ \ sq)) = \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \wedge \mathtt{WF}(sq) \\
\mathtt{WF}_s(\mathtt{not}\ \ F) = \mathtt{WF}_s(F) & \mathtt{WF}_s(p(\overrightarrow{e^a},\ \mathtt{any}\ \ sq)) = \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \wedge \mathtt{WF}(sq) \\
\mathtt{WF}_s(\mathtt{true}) = \mathit{true} & \mathtt{WF}_s(*\ \text{in}\ \ sq) = s = \mathit{sort}(sq)
\end{array}
$$

$$
\mathtt{WF}_s(\overrightarrow{e^a\ \mathtt{as}\ \alpha}\ \text{in}\ \ sq) = \bigwedge_{e^a} \mathcal{A}(e^a) \subseteq s \wedge \bigcup_{e^a} \{\alpha\} = \mathit{sort}(sq)
$$

$$
\begin{array}{ll}
\mathit{builtupon}(\mathit{value}, \overrightarrow{e^f}) = \mathit{true} & \\
\mathit{builtupon}(\alpha, \overrightarrow{e^f}) = \alpha \in \overrightarrow{e^f} & \mathit{builtupon}(f(\overrightarrow{e^a}), \overrightarrow{e^f}) = \bigwedge_{e^a} \mathit{builtupon}(e^a, \overrightarrow{e^f}) \\
& \mathit{builtupon}(f(\overrightarrow{e^f_1}), \overrightarrow{e^f}) = f(\overrightarrow{e^f_1}) \in e^f \vee (\bigwedge_{e^f_1} \mathit{builtupon}(e^f_1, \overrightarrow{e^f})) \\
\mathit{builtupon}(a(e^f_1), \overrightarrow{e^f}) = \mathit{true} &
\end{array}
$$

Fig. 7. Well-formedness

$\mathbf{SQL}_{\mathbf{Coq}}$ queries sorts The notion of *sort* is the $\mathrm{SQL}_{\mathrm{Coq}}$ counterpart of the notion of sorts in the relational model *i.e.*, a finite set of attributes. More precisely, following [1], we assume that the *set* of attribute names together with their corresponding types is *globally* defined. This implies that *typing* is handled by sorts: no two attributes with the same name and different types can exist. Sorts are recursively defined below. In order to define the base case, we assume that we are given a function *basesort*, which associates a set of attributes to each table name.

$$\begin{array}{l}
\mathit{sort}(\texttt{table}\ n) = \mathit{basesort}(n);\ \ \mathit{sort}(sq[\overrightarrow{b_i\ \texttt{as}\ a_i}]) = \bigcup_i\{a_i\};\ \ \mathit{sort}(sq[*]) = \mathit{sort}(sq) \\
\mathit{sort}(sq_1 \Box sq_2) = \mathit{sort}(sq_1),\ \text{where } \Box \in \{\ \texttt{union}\ ,\ \texttt{intersect}\ ,\ \texttt{except}\ \} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{and } \mathit{sort}(sq_1) = \mathit{sort}(sq_2) \\
\mathit{sort}(\texttt{select}\ \overrightarrow{e_i\ \texttt{as}\ a_i}\ \texttt{from}\ \overrightarrow{sq_j[r_j]}\ \ \texttt{where}\ F_1\ \texttt{group by}\ G\ \texttt{having}\ F_2) = \bigcup_i\{a_i\} \\
\mathit{sort}(\texttt{select}\ *\ \texttt{from}\ \overrightarrow{sq_j[r_j]}\ \ \texttt{where}\ F_1\ \texttt{group by}\ G\ \texttt{having}\ F_2) = \bigcup_j \mathit{sort}(sq_j[r_j])
\end{array}$$

Well-formed $\mathbf{SQL_{Coq}}$ queries The well-formedness condition serves different purposes. First, it ensures that sorts are sets. Second it guarantees that bag-theoretic (`union`, `intersect` and `except`) operators are sort compatible *i.e.*, that their arguments have the same sorts. Third, it prevents from having dangling attributes in the context. It discards queries that are rejected by SQL, hence, rejecting non linear queries. It imposes `from` clauses to be true Cartesian products by forcing attributes disambiguation. Last, it allows to discard queries that do not have an algebraic counterpart.

More precisely let us detail the definition given in Figure 7 step by step. The first two lines are straightforward. The definition for the `select-from-where-group-by-having` clause deserves some comments. Condition $\mathit{pairwise}_{\neq}(\overrightarrow{\alpha})$ forces tuples resulting from the evaluation of $\overrightarrow{e^a\ \texttt{as}\ \alpha}$ to have a support that is a set. Condition $\mathit{pairwise}_{\cap=\emptyset}(\overrightarrow{\mathit{sort}(sq_j[r_j])})$ states that the sorts of the from part are pairwise disjoint, and condition $\mathit{pairwise}_{\neq}(\overrightarrow{sq_j[r_j]})$ ensures that they are pairwise distinct, hence forcing the (evaluation of) `from` to be true Cartesian products and discarding non linear queries. Notation $\mathcal{A}(e^a)$ represents the set of attributes occurring in e^a. By imposing $\mathcal{A}(e^a) \subseteq s$, `WF` ensures that no dangling references to attributes in the (`select` $\overrightarrow{e^a\ \texttt{as}\ \alpha}$) are possible which is further achieved, thanks to $\texttt{WF}_s$, for attributes in the `where` or `having` clauses. $\texttt{WF}_s$ defines the well-formedness condition for formulae and consists in a classical structural inductive definition. In particular when the `where` condition is of the form $(\overrightarrow{e^a\ \texttt{as}\ \alpha})\ \texttt{in}\ sq$, $\texttt{WF}_s$ imposes the support of the left hand side to be equal to the sort of sq. The last condition, $\mathit{builtupon}(e^a, \overrightarrow{e^f})$ is more involved, informally this condition establishes that e^a is an expression only built from $\overrightarrow{e^f}$, constants and any aggregates $(a(e^f_1))$, thus, guaranteeing that the groups generated by the `group-by` have an homogeneous behaviour w.r.t., the evaluation of F_2 and the computation of the outermost `select`.

4.2 $\mathbf{SQL_{Coq}}$: semantics

We assume that we are given a database instance $\llbracket\_\rrbracket_{base}$ defined as a function from relation names to *bags* of tuples as well as $\llbracket\_\rrbracket_p$ an interpretation for each predicate symbol *i.e.*, a function from vectors of values to Booleans and $\llbracket\_\rrbracket_{ag}$ and $\llbracket\_\rrbracket_{fun}$ interpretations for symbols of aggregates and functions respectively. We denote by $t[\overrightarrow{a_i \texttt{as} b_i}]$ the tuple u defined by:

$$\begin{array}{l} support(u) = \{b_i\}_i \\ \forall i, u.b_i = t.a_i \end{array}$$

We then define $\llbracket\_\rrbracket_{\mathrm{SQL_{coq}}}$ the semantics of SQL queries, and give in Figure 8, $\llbracket\_\rrbracket_b$, the semantics of formulae. The basic cases, where $\cup$, $\cap$ and $\setminus$ correspond to the bag operators, are straightforward:

$\llbracket p(\overrightarrow{a_i})(t)\rrbracket_b = \llbracket p\rrbracket_p(\overrightarrow{t.a_i})$

$\llbracket p(\overrightarrow{a_i}, \texttt{all}\ sq)(t)\rrbracket_b$ is true iff forall tuple u in $\llbracket sq\rrbracket_{\mathrm{SQL_{coq}}}$, $\llbracket p\rrbracket_p(\overrightarrow{t.a_i}, u.sort(sq))$ holds

$\llbracket p(\overrightarrow{a_i}, \texttt{any}\ sq)(t)\rrbracket_b$ is true iff there exists a tuple u in $\llbracket sq\rrbracket_{\mathrm{SQL_{coq}}}$,
such that $\llbracket p\rrbracket_p(\overrightarrow{t.a_i}, u.sort(sq))$ holds

$\llbracket(*\ \texttt{in}\ sq)(t)\rrbracket_b$ is true iff t belongs to the set $\llbracket sq\rrbracket_{\mathrm{SQL_{coq}}}$

$\llbracket(\overrightarrow{a_i \texttt{as} b_i}\ \texttt{in}\ sq)(t)\rrbracket_b$ is true iff $t[\overrightarrow{a_i \texttt{as} b_i}]$ belongs to the set $\llbracket sq\rrbracket_{\mathrm{SQL_{coq}}}$

Fig. 8. Formulae semantics

$$\begin{array}{ll}
\llbracket \texttt{table}\ name\rrbracket_{\mathrm{SQL_{coq}}} & = \llbracket name\rrbracket_{base} \\
\llbracket sq\ \texttt{union}\ sq\rrbracket_{\mathrm{SQL_{coq}}} & = \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}} \cup \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}} \\
\llbracket sq\ \texttt{intersect}\ sq\rrbracket_{\mathrm{SQL_{coq}}} & = \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}} \cap \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}} \\
\llbracket sq\ \texttt{except}\ sq\rrbracket_{\mathrm{SQL_{coq}}} & = \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}} \setminus \llbracket sq\rrbracket_{\mathrm{SQL_{coq}}}
\end{array}$$

The most complex case is the `select-from-where-groupby-having` one. Informally, a first step consist in evaluating the `from` and `where` parts. Then the (intermediate) collection of tuples obtained is partitioned according to the `group-by` criteria yielding a collection of collections of tuples. Each such collection being homogeneous w.r.t., the grouping criteria and the `having` condition. Last, the `select` clause is applied yielding again a collection of tuples as a result. More formally:

$$\llbracket \texttt{select}\ e^a \texttt{as}\ \alpha\ \texttt{from}\ \overrightarrow{sq[r]}\ \texttt{where}\ F_1\ \texttt{group by}\ G\ \texttt{having}\ F_2\rrbracket_{\mathrm{SQL_{coq}}}$$

$$= \quad \left\{ T[\overrightarrow{e^a \texttt{as}\ \alpha}] \;\middle|\; \begin{array}{l} \llbracket F_2\rrbracket_b(T) = true \wedge \\ T \in \texttt{partition}(G, \llbracket \texttt{select}\ *\ \texttt{from}\ \overrightarrow{sq[r]}\ \texttt{where}\ F_1\rrbracket_{\mathrm{SQL_{coq}}}) \end{array} \right\}$$

where

$$T[\overrightarrow{e_{a_i}\texttt{as}b_i}] = \begin{cases} \text{support} = \bigcup_i \{b_i\} \\ T[\overrightarrow{e^a\texttt{as}b_i}](b_i) = T(e^a) \end{cases}$$

$T(e^a)$ being defined by:

$$T(e^a) = \begin{cases} \text{if } e^a = e^f\text{, then } e^f(t), \text{for any } t \in T \\ \text{if } e^a = f(\overrightarrow{e_{a_i}})\text{, then } \llbracket f \rrbracket_{fun}(\overrightarrow{e_{a_i}(T)}) \\ \text{if } e^a = a(e)\text{, then } \llbracket a \rrbracket_{ag}\{e(t) \mid t \in T\} \end{cases}$$

provided that T is a non-empty collection of tuples, homogeneous w.r.t., aggregate e^a. and where `partition` is defined by:

$$\texttt{partition}(\overrightarrow{e^f}, \mathcal{S}) = \bigcup_{t \in \mathcal{S}} \{\{s \in \mathcal{S} \mid \forall e^f_i \in \overrightarrow{e^f}, s(e^f_i) = t(e^f_i)\}\}$$

5 Extended algebra

As illustrated in Section 3 relational algebra cannot capture SQL queries. In this section we present `ExtAlg` a very concise, yet expressive, *bag-set* algebra together with its (mechanised) semantics $\llbracket\_\rrbracket_{\mathrm{ExtALG}}$. `ExtAlg`, non trivially, extends the SPJR algebra presented in Section 2 allowing us to relate $\mathrm{SQL}_{\mathrm{Coq}}$ semantics, thus SQL's one, to it as will be shown in Section 6. Our Coq formalisation is based on, borrows and extends, the work in [4]. We then formally prove using Coq the correctness of the embedding of the SPJR algebra (taken from [4]) into `ExtAlg`.

5.1 A concise extended algebra

As we wanted `ExtAlg` to be extensible and to acknowledge the relational algebra, it hosts sets and bags through the general type of collection. The syntax of `ExtAlg` is given by:

$$\begin{array}{lll}
q & ::= & \emptyset \mid \{()\} \mid r \mid \omega_{P,F,c}(q) \mid q \bowtie q \mid q \cup q \mid q \cap q \mid q \setminus q \\
P & ::= & \texttt{fine} \mid \texttt{partition}(\overrightarrow{e^{x\cdot f}}) \\
F & ::= & \top \mid p(\overrightarrow{e^{x\cdot a}}) \mid F \vee F \mid F \wedge F \mid \neg F \mid \forall x F \mid \exists x F \\
c & ::= & \overrightarrow{\texttt{code}(\alpha, e^a)} \\
x & ::= & \texttt{var}\ q\ nat \\
e^{x\cdot f} & ::= & value \mid x.\alpha \mid f(\overrightarrow{e^{x\cdot f}}) \\
e^{x\cdot a} & ::= & e^{x\cdot f} \mid a(e^{x\cdot f}) \mid f(\overrightarrow{e^{x\cdot a}}) \\
p & ::= & < \mid > \mid \leq \mid \geq \mid \ldots \mid \textit{user defined predicate} \\
f & ::= & + \mid - \mid * \mid \ldots \mid \textit{user defined function} \\
a & ::= & \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Avg} \mid \textit{user defined aggregate}
\end{array}$$

The empty collection of tuples ($\emptyset$), the singleton containing the empty tuple ($\{()\}$)[12] and relation's names (r) are algebraic expressions with intended obvious meaning. The core of `ExtAlg` consists in two operators: the SPJR natural join ($\bowtie$) and a new operator ω which takes as operand a query q and three parameters: P a partition criteria, F a formula and c a sequence of pairs (attribute names, and expressions e^a). Notice that `code` embeds e^a's, as they were defined in Section 4, and not $e^{x.a}$. This is relevant since in a context where an expression contains a single free variable and no bounded variables (as it will be clear when expliciting ω associated semantics) this free variable could be left implicit. Let us illustrate the versatility of `ExtAlg`, considering the following $\mathrm{SQL}_{\mathrm{Coq}}$ queries:

```
let rho0 := a as a0, b as b0, c as c0 in
let rho1 := a as a1, b as b1, c as c1 in

select * from from tbl1[*]
where (a+b) >= all (select (a0 + c1) as a0_plus_c1 from tbl0[rho0], tbl1[rho1]);
```

which is expressed as:

$$\begin{array}{l}
let\ \ tbl'_1 := tbl_1 \bowtie \mathtt{Empty\_Tuple}\ \ in\\
let\ \ tbl''_1 := \omega_{\mathtt{fine},\top,\mathtt{id}}(tbl'_1)\ \ in\\
let\ \ q_{01} := (tbl_0[\rho_0] \bowtie tbl_1[\rho_1]) \cap (tbl_0[\rho_0] \bowtie tbl_1[\rho_1])\ \ in\\
let\ \ q_i := \omega_{\mathtt{fine},\top,\mathtt{code}(a_0\_plus\_c_1,a0+c1)}(q_{01})\ \ in\\
let\ \ F := \forall x_{q_i}, x.a + x.b \geq x_{q_i}.(a_0\_plus\_c_1)\ \ in\\
\omega_{\mathtt{fine},F,\mathtt{id}}(tbl'_1) \cap tbl''_1
\end{array}$$

and

```
select a as a, count(b) as countb from tbl1[*]
group by a having avg(c) >= all (select a as a from tbl1[*]);
```

expressed by:

$$\begin{array}{l}
let\ \ tbl'_1 := tbl_1 \bowtie \mathtt{Empty\_Tuple}\ \ in\\
let\ \ tbl''_1 := \omega_{\mathtt{fine},\top,\mathtt{id}}(tbl'_1)\ \ in\\
let\ \ q_i := \omega_{\mathtt{fine},\top,\mathtt{code}(a,a)}(tbl1'' \cap tbl1'')\ \ in\\
let\ \ F := \forall x_{q_i}, x.\alpha^{+2}_{def} \geq x_{q_i}.a\ \ in\\
\omega_{\substack{\mathtt{fine},\top,\\ \mathtt{code}(a,a')\\ \mathtt{code}(count_b,count'_b)}}\left(\omega_{\substack{\mathtt{partition}\{\mathtt{a}\},F,\\ \mathtt{code}(\alpha^{+2}_{def},avg(c))\\ \mathtt{code}(count'_b,count(b))\\ \mathtt{code}(a,a)\\ \mathtt{code}(b,b)\\ \mathtt{code}(c,c)\\ \mathtt{code}(a',a)}}(tbl1'')\right)
\end{array}$$

[12] More precisely, it is a family of such singletons indexed by the relation's sort.

Operator ω has the following semantics:

$$\llbracket \omega_{P,F,\{\mathtt{code}(a_i,c_i)\}_i}(q) \rrbracket_{\mathrm{ExtALG}} =$$

$$\left\{ t \,\middle|\, \begin{array}{l} \mathit{support}(t) = \{a_i\}_i \;\; \wedge \\ \exists T \in \llbracket P \rrbracket_{\mathrm{ExtALG}}(\llbracket q \rrbracket_{\mathrm{ExtALG}}),\;\; t.a_i = \llbracket c_i \rrbracket_{\mathrm{ExtALG}}(T) \wedge \llbracket F \rrbracket_{\mathrm{ExtALG}}(t) = \mathit{true} \end{array} \right\}$$

We do not detail interpretation of formulae nor do we detail interpretation of functions and aggregates. The only point to mention is that bounded variables in formulae will be interpreted as tuples in $\llbracket q \rrbracket_{\mathrm{ExtALG}}$. We refer the reader to appendix **??** for the whole (Coq) definition. Operator ω, allows for capturing renamings, aggregates, functions and `group-by-having` as will be formally established in Section 6.

5.2 Embedding SPJR into `ExtAlg`

`ExtAlg` also hosts the SPJR algebra. More precisely the embedding $\mathcal{E}$ of SPJR into `ExtAlg` is defined by:

$$\begin{array}{ll} \mathcal{E}(r) & = r \\ \mathcal{E}(\pi_W(q)) & = \omega_{\mathtt{fine},\top,\mathtt{id}(W)}(\mathcal{E}(q)) \\ \mathcal{E}(\sigma_F(q)) & = \omega_{\mathtt{fine},\mathcal{E}(F),\mathtt{id}(\mathit{sort}(q))}(\mathcal{E}(q)) \\ \mathcal{E}(\rho(q)) & = \omega_{\mathtt{fine},\top,\{\mathtt{code}(\rho(a_i),a_i)\}_{a_i \in \mathit{sort}(q)}}(\mathcal{E}(q)) \\ \mathcal{E}(q_1 \bowtie q_2) & = \mathcal{E}(q_1) \bowtie \mathcal{E}(q_2) \\ \mathcal{E}(q_1 \,\square\, q_2) & = \mathcal{E}(q_1) \,\square\, \mathcal{E}(q_2) \end{array}$$

where

$$\mathtt{id}(W) = \{\mathtt{code}(a,a)\}_{a \in W}$$

and all operators in `ExtAlg` are tagged by the `set` flag. $\mathcal{E}(F)$ is formally given in the Coq definition of algebra_to_ealgebra and simply consist in structurally applying the embedding.

There is a last subtle point worth to mention. As the reader could notice:

$$\mathcal{E}(\pi_{\mathit{sort}(q)}(q)) = \mathcal{E}(\rho_{\mathit{id}_{\mathit{sort}(q)}}(q))$$

However, we want $\mathcal{E}$ to preserve that two syntactically different SPJR-algebraic queries differs in `ExtAlg`. Therefore a first stage consists in normalising the SPJR query based on the following rewriting rules:

$$\begin{array}{ll} \mathcal{N}(\sigma_{\mathit{true}}(q)) & \rightsquigarrow q \\ \mathcal{N}(\pi_{\mathit{sort}(q)}(q)) & \rightsquigarrow q \\ \mathcal{N}(\rho_{\mathit{id}}(q)) & \rightsquigarrow q \end{array}$$

We are able to state the embedding's correctness theorem whose Coq counterpart is given in Appendix **??**.

Theorem 1. *Let q be a well-formed SPJR query, for any well-sorted instance, $[\![q]\!]_{SPJR} = [\![\mathcal{E}(\mathcal{N}(q))]\!]_{ExtALG}$.*

The main difficulty encountered in proving the theorem was to formally establish that $\mathcal{N}$ was indeed preserving the semantics of the query. This was delicate because we have explicit variables in formulae and that in relational algebra at most one free variable may occur in a formula.

5.3 Query logical optimisation: `ExtAlg` rewritings

Main classical rewritings proven in [4] are transported in the context of `ExtAlg`.

$$
\begin{array}{rlrl}
\sigma_{f_1 \wedge f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) & (1) & \pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) & \text{if } W_1 \subseteq W_2 \ (5) \\
\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q)) & (2) & \pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q)) & \text{if } \mathcal{A}tt(f) \subseteq W \ (6) \\
(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3) & (3) & \sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 & \text{if } \mathcal{A}tt(f) \subseteq sort(q_1) \ (7) \\
q_1 \bowtie q_2 \equiv q_2 \bowtie q_1 & (4) & \sigma_f(q_1 \square q_2) \equiv \sigma_f(q_1) \square \sigma_f(q_2) & \text{where } \square \text{ is } \cup \text{ or } \cap \ (8)
\end{array}
$$

The proofs were not involved and each of them took around 150loc. Notice that they take into account the fact that membership is achieved modulo *tuple equivalence* and not with *tuple syntactic Leibniz equality.*

Less classical rewritings are based on the θ-join operator ($\bowtie_\theta$) which is defined by $\sigma_\theta(q_1 \times q_2)$, and on the θ-semi-join ($\ltimes_\theta$) which is a derived bag algebra operator that preserves the multiplicity of tuples. It is, informally, expressed as $q_1 \ltimes_\theta q_2 =_{\texttt{def}} (q_1 \bowtie (\delta(\pi_{sort(q_1)}(q_1 \bowtie_\theta q_2)))$ where δ stands for duplicate elimination. In our context operator δ is derived from our primitive operators as

$$\delta(q) = \omega_{\texttt{partition}(sort(\texttt{q})),\ \top,\ \{\texttt{code}(a,a)\}_{a \in sort(q)}}(q)$$

We proved the equivalences θ-semi-join introduction and θ-semi-join push expressed in [15].

$$
\begin{array}{lll}
q_1 \bowtie_\theta q_2 & \equiv q_1 \bowtie_\theta (q_2 \ltimes_\theta q_1) & (9) \\
(q_1 \bowtie_{\theta_1} q_2) \ltimes_{\theta_2} q_3 & \equiv (q_1 \bowtie_{\theta_1} q_2') \ltimes_{\theta_2} q_3 & (10) \\
 & \text{where } q_2' \text{ stands for } q_2 \ltimes_{\theta_1 \wedge \theta_2} (q_1 \times q_3)) &
\end{array}
$$

This strengthen our conviction that `ExtAlg` is adapted for hosting data-centric languages. In future work, based on [4] in which we modelled integrity constraints (functional and general dependencies) we shall prove more equivalences that do exploit such dependencies.

6 A Coq mechanised SQL's compilation chain

As explained in the introduction, SQL compilers proceed in four steps corresponding to two phases: the parsing and the planning. The first two steps translate SQL queries into abstract syntax trees whose nodes are, in theory, algebraic operators and whose leaves are relations. We rather produce an extended algebra expression. Its definition is given in Figure 9. Obviously it is very involved

$\mathcal{T}(\texttt{table}\ name) = name \qquad \mathcal{T}(sq_1\ \texttt{union}\ sq_2) = \mathcal{T}(sq_1) \cup \mathcal{T}(sq_2)$

$\mathcal{T}(sq_1\ \texttt{intersect}\ sq_2) = \mathcal{T}(sq_1) \cap \mathcal{T}(sq_2) \qquad \mathcal{T}(sq_1\ \texttt{except}\ sq_2) = \mathcal{T}(sq_1) \setminus \mathcal{T}(sq_2)$

$\mathcal{T}(\texttt{select}\ *\ \texttt{from}\ \overrightarrow{sq[r]}\ \ \texttt{where}\ F_1\ \texttt{group by}\ G\ \texttt{having}\ F_2) =$

$\quad let\ \ \overrightarrow{s} := \overrightarrow{\alpha_i\ \texttt{as}\ \alpha_i}, \alpha_i \in \bigcup \overrightarrow{sort(sq[r])}\ \ in$

$\quad \mathcal{T}(\texttt{select}\ \overrightarrow{s}\ \texttt{from}\ \overrightarrow{sq[r]}\ \ \texttt{where}\ F_1\ \texttt{group by}\ G\ \texttt{having}\ F_2)$ (desuggaring)

$\mathcal{T}(\texttt{select}\ \overrightarrow{e^a\ \texttt{as}\ \alpha}\ \texttt{from}\ \overrightarrow{sq[r]}\ \ \texttt{where}\ F_1\ \texttt{group by singleton having}\ F_2) =$

$\quad let\ \ q_1 := \mathcal{T}_{\texttt{from}}\,(\overrightarrow{sq[r]})\ \ in$

$\quad \omega_{\texttt{fine},\top,\overrightarrow{\texttt{code}(\alpha,e^a)}}(\mathcal{T}_F\,(\texttt{fine}, \texttt{id}(sort(q_1)), q_1, F_1\ \ \texttt{and}\ \ F_2))$

$\mathcal{T}(\texttt{select}\ \overrightarrow{e_i^a\ \texttt{as}\ \alpha_i}\ \texttt{from}\ \overrightarrow{sq[r]}\ \ \texttt{where}\ F_1\ \texttt{group by}\ \overrightarrow{e^f}\ \texttt{having}\ F_2) =$

$\quad let\ \ q_1 := \mathcal{T}_{\texttt{from}}\,(\overrightarrow{sq[r]})\ \ in$

$\quad let\ \ q_2 := \mathcal{T}_F\,(\texttt{fine}, \texttt{id}(sort(q_1)), q_1, F_1)\ \ in$

$\quad let\ \ m_2\ \ be\ 1 +$ the maximum of indexes occurring in the attributes of $sort(q_2)\ \ in$

$\quad let\ \ m_3\ \ be\ (1 + m_2) +$ the maximum of the indexes occurring in the attributes of $\overrightarrow{e_i^a}\ \ in$

$\quad let\ \ la := \{(\alpha_{def}^{+m_3+j}, e_j^a) \mid \{e_j^a\} = \mathcal{E}xp(F_2)\}\ \ in$

$\omega_{\texttt{fine},\top,\overrightarrow{\texttt{code}(\alpha_i,\alpha_i^{+m_2})}}(\mathcal{T}_F(\overrightarrow{e^f}, \texttt{id}(sort(q_2)) \cup \overrightarrow{\texttt{code}(la)} \cup \overrightarrow{\texttt{code}(\alpha_i^{+m_2}, e_i^a)}_i, q_2, F_2^{+la}))$

$\quad$ where $\alpha_{def}^{+m_3+j}$ is a default attribute, shifted in order to avoid capture,

$\quad$ and F_2^{+la} is the result of applying the corresponding substitution la to formula F_2

$\mathcal{T}_{\texttt{from}}\,\overrightarrow{(sq_j[r_j])} = \bowtie_j \mathcal{T}_\rho\,(sq_j[r_j])$

$\mathcal{T}_\rho\,(sq[*]) = \mathcal{T}\,(sq) \qquad \mathcal{T}_\rho\,(sq[\overrightarrow{\beta_i\ \texttt{as}\ \alpha_i}]) = \omega_{\texttt{fine},\top,\overrightarrow{\texttt{code}(\alpha_i,\beta_i)}}(\mathcal{T}\,(sq))$

$\mathcal{E}xp(true) = \emptyset \qquad \mathcal{E}xp(\texttt{not}\ \ F) = \mathcal{E}xp(F)$

$\mathcal{E}xp(F_1\ \ \texttt{and}\ \ F_2) = \mathcal{E}xp(F_1\ \ \texttt{or}\ \ F_2) = \mathcal{E}xp(F_1)\ \cup\ \mathcal{E}xp(F_2)$

$\mathcal{E}xp(p(\overrightarrow{e^a})) = \mathcal{E}xp(p(\overrightarrow{e^a},\ \texttt{all}\ sq)) = \mathcal{E}xp(p(\overrightarrow{e^a},\ \texttt{any}\ sq)) = \bigcup_{e^a}\{e^a\}$

$\mathcal{E}xp(*\ \text{in}\ \ sq) = \bigcup_{\alpha \in sort(sq)}\{\alpha\} \qquad \mathcal{E}xp(\overrightarrow{e^a\ \texttt{as}\ \alpha}\ \text{in}\ \ sq) = \bigcup_{e^a}\{e^a\}$

$\mathcal{T}_V(q, n, f(\overrightarrow{e^a})) = f(\overrightarrow{\mathcal{T}_V(q, n, e^a)}) \qquad \mathcal{T}_V(q, n, f(\overrightarrow{e^f})) = f(\overrightarrow{\mathcal{T}_V(q, n, e^f)})$

$\mathcal{T}_V(q, n, a(e^f)) = a(\mathcal{T}_V(q, n, e^f)) \qquad \mathcal{T}_V(q, n, \alpha) = (\texttt{var}\ q\ n).\alpha$

$\mathcal{T}_V(q, n, value) = value$

$\mathcal{T}_F\,(G, c, q, true) = \omega_{G,\top,c}(q) \qquad \mathcal{T}_F\,(G, c, q, \texttt{not}\ \ F) = \omega_{G,\top,c}(q) \setminus \mathcal{T}_F\,(G, c, q, F)$

$\mathcal{T}_F\,(G, c, q, F_1\ \ \texttt{and}\ \ F_2) = \mathcal{T}_F\, G, c, q, F_1)\ \cap\ \mathcal{T}_F\,(G, c, q, F_2)$

$\mathcal{T}_F\,(G, c, q, F_1\ \ \texttt{or}\ \ F_2) = (\mathcal{T}_F\,(G, c, q, F_1)\ \cup\ \mathcal{T}_F\,(G, c, q, F_2)) \setminus (\mathcal{T}_F\, G, c, q, F_1)\ \cap\ \mathcal{T}_F\,(G, c, q, F_2))$

$\mathcal{T}_F\,(G, c, q, p(\overrightarrow{e^a})) = \omega_{G,c,p(\overrightarrow{\mathcal{T}_V(q,\ 0,\ e^a)})}(q)$

$\mathcal{T}_F\,(G, c, q, p(\overrightarrow{e^a, \texttt{all}\ \ sq})) = let\ \ x_{sq} := \texttt{var}(\mathcal{T}\,(sq), 1)\ \ in\ \ \omega_{G,\forall x_{sq},\ \ p(\overrightarrow{\mathcal{T}_V(q,\ 0,e^a)},\overrightarrow{x_{sq}.a}_{a \in sort(sq)})),c}(q)$

$\mathcal{T}_F\,(G, c, q, p(\overrightarrow{e^a, \texttt{any}\ \ sq})) = \omega_{G,\exists(\texttt{var}(\mathcal{T}\,(sq),1),\ \ p(\overrightarrow{\mathcal{T}_V(q,\ 0,e^a)},\overrightarrow{\mathcal{T}_V(\mathcal{T}\,(sq),1,a)}_{a \in sort(sq)})),c}(q)$

$\mathcal{T}_F\,(G, c, q, *\ \text{in}\ \ sq) = \omega_{G,c,\top}(q)\ \ \bowtie\ \ \delta(\omega_{G,\top,c}(\mathcal{T}\,(sq)))$

$\mathcal{T}_F\,(G, \overrightarrow{\texttt{code}(\alpha_i, e_i^a)}, q, \overrightarrow{e^a\ \texttt{as}\ \alpha}\ \text{in}\ \ sq) =$

$\quad let\ \ q_G := \omega_{G,\top,\overrightarrow{\texttt{code}(\alpha_i,e_i^a)}}(q)\ \ in$

$\quad let\ m_1\ be\ 1 +$ the maximum of indexes occurring in the attributes of $sort(q_G)\ \ in$

$\quad let\ \ q' := q_G \bowtie \delta(\mathcal{T}(sq)^{+m_1})\ \ in\ \ let\ \ F := \bigwedge_{e^a\ \texttt{as}\ \alpha}(var\ q'\ 0).\alpha = \mathcal{T}_V(q', 0, e^a)\ \ in$

$\omega_{\texttt{fine},\top,\texttt{id}(sort(q_G))}(\omega_{\texttt{fine},F,\texttt{id}(sort(q'))}(q'))$

Fig. 9. SQL syntactic and semantics steps

and deserves some comments. The first four cases are straightforward. The most complex cases are the `select` ones. Let us recall the evaluation order of $\mathrm{SQL}_{\mathrm{Coq}}$ and `ExtAlg` respectively. Consider query: `select s from lsq where F1 group by G having F2`. $\mathrm{SQL}_{\mathrm{Coq}}$ and SQL first evaluate the `from` part `lsq` and filter the resulting *collection of tuples* w.r.t., `F1` in a second step they build groups thanks to `G` yielding a *collection of collections of tuples* which is further filtered by `F2`. At the end the remaining collections of tuples are *flattened* using the `select` part.

`ExtAlg` proceeds slightly differently. First it builds a *collection of collections of tuples* according to its partition's criterion P, then it *flattens* the collection by evaluating the `code` part, and filters with its formula F the resulting tuples. Notice that there is a discrepancy between both evaluation's orderings. The rationale for such a discrepancy lies in many aspects. First, in our wish that `ExtAlg` be as concise as possible, thus minimising the number of operators. It also lies in the fact that this development is part of a more general library embedding standard first-order logic. Hence formulae deal with individuals rather than sets, bags or any type of collections. Last and not least, for the sake of generality, we wanted `ExtAlg` to be *data-model agnostic.*

The consequence is that for parsing the `having` condition we have to build a formula F_2^{+la} that behaves as F_2 but also, we have to simulate each group, s by an individual tuple t such that: $F_2(s) = F_2^{+la}(t)$. Moreover, each group filtered by F_2 must yield a tuple obtained thanks to the `select` part of the query. Hence, tuple t is built from an arbitrary element of s (thanks to the homogeneity hypothesis imposed by `WF`), the expressions freely occurring in F_2 ($\mathcal{E}xp(F_2)$) and the expressions e_i^a occurring in the `select` part $\overrightarrow{e_i^a \texttt{ as } \alpha_i}$. This is captured by: $\texttt{id}(sort(q_2)) \cup \overrightarrow{\texttt{code}(la)} \cup \overrightarrow{\texttt{code}(\alpha_i^{+m_2}, e_i^a)}_i$.

In order to avoid overlapping between the three parts of t we shifted the corresponding attributes. This is expressed by the superscript notation α^{+j} and α_{def}^{+j} where j represents an offset and α_{def} a default attribute name.

The last, subtle, aspect to be detailed is the treatment of $\overrightarrow{e^a \texttt{ as } \alpha}$ in sq. First, let us explain the intuitive meaning of:

$$\mathcal{T}_F(G, \overrightarrow{\texttt{code}(\alpha_i, e_i^a)}, q, F)$$

It should result in an algebraic expression, which when interpreted, exactly contains the tuples t, with the same number of occurrences, built from a group in $[\![q]\!]_{\mathrm{ExtALG}}$ partitioned according to G, evaluated by the `code` part $\overrightarrow{\texttt{code}(\alpha_i, e_i^a)}$ and satisfying F. The first conditions are expressed by $t \in [\![q_G]\!]_{\mathrm{ExtALG}}$ where $q_G = \omega_{G, \top, \overrightarrow{\texttt{code}(\alpha_i, e_i^a)}}(q)$. Let t be such a tuple, it then fulfils the above `in` condition when $t[\overrightarrow{e^a \texttt{ as } \alpha}]$ belongs to $[\![sq]\!]_{\mathrm{SQL_{coq}}}$, and, provided that the parser is sound, also to $[\![\mathcal{T}(sq)]\!]_{\mathrm{ExtALG}}$ which is equivalent to:

$$(t, t[\overrightarrow{e^a \texttt{ as } \alpha}]) \in [\![q_G \bowtie \delta(\mathcal{T}(sq))]\!]_{\mathrm{ExtALG}}$$

Notice that δ is used on the right part of the natural join, in order to keep the multiplicity of t. Another way to express this is:

$$(t, t') \in [\![\omega_{\mathtt{fine}, F', \mathtt{id}}(q_G \bowtie \delta(\mathcal{T}(sq)))]\!]_{\mathrm{ExtALG}}$$

where F' expresses that t' is actually equal to $t[\overrightarrow{e^a \ \mathtt{as} \ \alpha}]$:

$$F' = \bigwedge_{e^a \ \mathtt{as} \ \alpha} x_{sq}.a = e^a(x_{q_G})$$

By using the appropriate offset $^{+m_1}$ over $\mathcal{T}(sq)$, one can ensure that q_G and $\mathcal{T}(sq)$ do not interfere in an another way than by F', which leads to the actual formulation:

$$\begin{array}{l}
\mathcal{T}_F\,(G, \overrightarrow{\mathtt{code}(\alpha_i, e^a_i)}, q, \overrightarrow{e^a \ \mathtt{as} \ \alpha} \text{ in } \ sq) = \\
\qquad \mathit{let} \ \ q_G := \omega_{G, \top, \overrightarrow{\mathtt{code}(\alpha_i, e^a_i)}}(q) \ \ \mathit{in} \\
\mathit{let} \ \ q' := q_G \bowtie \delta(\mathcal{T}(sq)^{+m_1}) \ \ \mathit{in} \\
\mathit{let} \ \ F' := \bigwedge_{e^a \ \mathtt{as} \ \alpha}(\mathit{var} \ q' \ 0).\alpha = \mathcal{T}_V(q', 0, e^a) \ \ \mathit{in} \\
\omega_{\mathtt{fine}, \top, \mathtt{id}(\mathit{sort}(q_G))}(\omega_{\mathtt{fine}, F', \mathtt{id}(\mathit{sort}(q'))}(q'))
\end{array}$$

Now we are able to state the adequation theorem:

Theorem 2. *Let sq be a well-formed SQL query. Then for any well-sorted instance,*

$$[\![sq]\!]_{SQL_{coq}} = [\![\mathcal{T}(sq)]\!]_{ExtALG}$$

$\mathcal{T}$ has been written in Gallina and we formally proved its correctness. We then extracted its corresponding Ocaml, correct by construction, implementation.

Clearly, the proof of the adequation theorem was involved but enlightning. Indeed, very subtle aspects were raised thanks to Coq. For instance, it happened that the θ-semi-join and the δ duplicate elimination operators of Section 5 appeared essential for correctly translating the `in` SQL's predicate. More technically, translating the `in` consists in performing query decorrelation as presented in [16]. It took many efforts, over years, for the database community to correctly define query decorrelation. Such a (rewriting) technique is closely related to the notion of semi-joins and duplicate elimination. This did not escape Coq's attention!

7 Related work, conclusion and perspectives

7.1 Related Work

Many attempts have been made by the database community to define a formal semantics for SQL. Among those a first, realistic at that time, proposal can be found in [6]. The most significant work on the topic can be found in [14], were the authors addressed a credible subset of SQL (with no functions symbols and no nested queries though). In any case, none of those works did formally obtain strong guarantees as we did in this article, nor did they formally relate their proposed semantics with a deeply formalised algebra.

The first attempt to formalise the relational data model is found in [10, 9]. However, only the unnamed perspective is formalised using the Agda proof assistant. The first complete Coq formalisation of the relational model is found in [4] were the data model, the algebra as well as the integrity constraints aspects were modelled.

Many efforts to use proof assistants to mechanise commercial languages' semantics have been already done with the seminal work on Compcert [12] and later the work on JavaScript [5]. Recently, a similar approach as the one presented in this article, consisting in relating a language to an algebra, is undertaken by [17], in the context of an abstract pattern-calculus for rule languages intended to capture the essence of IBM's JRules. However only the algebra is mechanised and no semantics' preservation theorem is proven in this context.

The very first Coq formalisation of a SQL parser is found in Malecha *et al.*, [13]. However, they considered a very restricted subset of the language (with no group-by having clause, no aggregates). Moreover, probably for the sake of simplicity, they placed themselves in the context of an unnamed version of the language, in which attributes names are not denotable. Such a choice is a little unrealistic as, in standard SQL, attributes are denotable entities at the language level. Many of the most difficult problems arise when dealing with attributes as we illustrated in Section 3.

7.2 Conclusions

Contributions In this article we presented SQLCert a formal framework inherently based on attribute names which is the first executable mechanisation of SQL's semantics compliant with the theoretical algebraic foundations of the (relational) model of data. We non trivially extended our previous work [4] so as to deal with SQL. We proposed and formalised an extended *bag-set* algebra gracefully hosting the relational one and allowing the underlying database system to exploit well-known database optimisation techniques thus yielding trully certified rewritings commonly used by practical optimisers. We formalised an *embedding* of (the named) relational algebra into our extended algebra and proved its correctness. Unlike what is found in the literature, our (extended) algebra is very concise and more importantly is *data model agnostic*. We also provided a Coq mechanisation of the first three steps of the compiler, *formally* relating $\mathrm{SQL}_{\mathrm{Coq}}$ to its algebraic counterpart together with its Coq adequation proof and its Ocaml extraction. SQLCert is, to our knowledge, the *first* proposal of a (*realistic fragment of*) SQL compiler able to cope with *attributes' names* in a clean way, with *finite bags* thus *formally reconciling* SQL with the *algebraic foundations* of the *relational model and databases.*

Lessons We learnt a lot on the Coq, database as well as programming language design sides. Indeed, in order to capture SQL's specificity, we had to extend the relational data model and algebra presented in our previous work [4]. In an early version of the development, we defined `ExtAlg` with a pure set-theoretic semantics and only addressed the SQL's fragment with no duplicates. Then we addressed the multiset aspects of SQL. Doing so we were pleasantly surprised to discover that it was not so dramatic: the development, based on second author's existing work [7], went smoothly and it took us less than one month to account for multisets. Therefore, the widespread belief[13] that *the* problem for SQL's semantics is to assign it a bag semantics is not as crucial as it seemed to be. Moreover, going from sets to multisets allowed us to precisely pinpoint which aspects were relevant. For instance counting tuples' occurences was crucial for correctly translating the `in` predicate as well as for translating disjunction in formulae. Obviously the proof of the semantics' preservation theorem was, as expected, the most involving part of the development. Also, accurately and faithfully grasp SQL's semantics as described in the ISO/IEC document was painful. Even if we knew it, it confirmed us that SQL having initially been designed as a domain specific language intended *not* to be Turing-complete more features have been added to it along the time in the standardisation process, hence, seriously, and sadly, departing it from its original elegant foundations. This, definitely, made our mechanisation task more complex. However, SQL is the real-life (relational) database programming language and there is no way around it!

7.3 Perspectives

In the very short term, we shall include `NULL` values and order-based SQL's features. As we said previously, based on our experience of adding multisets in our development, we are confident that adding a new kind of collection, lists for instance, should not be as difficult as one could imagine. The next step to be addressed is to deeply specify the last part of the query compiler. Rather than mechanising the cost-based plan selection step, which seems far beyond what could be expected from Coq, we shall verify that whatever the plan is, it is correct w.r.t., its algebraic specification. A relevant approach could be to rely on the work presented in [2, 3]. In this line of research, the idea consists in implementing and specifying in Gallina/Coq the classical algorithms corresponding to relational algebra operators, implement them in C and then rely on the VST tool to manually prove that the C version is a correct refinement w.r.t., the specification. Equally valid could be to rely on the Why(3) deductive verification tool chain [8].

Last, our extended algebra is parametric w.r.t., the data model. No strong assumptions were made on the intrinsic nature of tuples and, we think, it is versatile enough to handle nested tuples and/or tree structured data. Indeed, we can assign to their respective associated accessors a predefined semantics thus opening the way for taking into account NoSQL languages in a clean, very concise,

[13] At least in the database community.

algebra-based framework, in sharp contrast with the many various, nested relational algebras existing in the literature. A first step towards this line of research will consist in taking the formalisation of [17] and embed it into `ExtAlg`.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] A. W. Appel. Verified software toolchain - (invited talk). In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011*, pages 1–17, 2011.

[3] A. W. Appel. *Program Logics - for Certified Compilers.* Cambridge University Press, 2014.

[4] V. Benzaken, E. Contejean, and S. Dumbrava. A Coq Formalization of the Relational Data Model. In *23rd European Symposium on Programming (ESOP)*, 2014.

[5] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014.

[6] S. Ceri and G. Gotlob. Translating SQL into relational algebra: Optimisation, semantics, and equivalence of SQL queries. *IEEE Trans., on Software Engineering*, SE-11:324–345, April 1985.

[7] E. Contejean. *Coccinelle: a Coq library for term rewriting.* `https://www.lri.fr/~contejea/Coccinelle/coccinelle.html`,.

[8] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[9] C. Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *LNCS*, pages 137–148. Springer, 2003.

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

[11] ISO/IEC. Information technology - database languages - SQL - part 2: Foundation (SQL/foundation), 2006. Final Commitee Draft.

[12] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[13] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[14] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3):513–534, 1991.

[15] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc., of the 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada, 1996.*, pages 435–446, 1996.

[16] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 450–458, 1996.

[17] A. Shinnar, J. Siméon, and M. Hirzel. A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 542–567, 2015.

[18] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2010. `http://coq.inria.fr`.

[19] The Isabelle Development Team. *The Isabelle Interactive Theorem Prover*, 2010. `https://isabelle.in.tum.de/`.