A Coq Mechanised Formal Semantics for Realistic SQL Queries

Véronique Benzaken, Évelyne Contejean

A Coq Mechanised Formal Semantics for Realistic SQL Queries*

Formally Reconciling SQL and Bag Relational Algebra

Véronique Benzaken
Université Paris Sud, LRI, France
veronique.benzaken@lri.fr

Évelyne Contejean
CNRS, Université Paris Sud, LRI, France
evelyne.contejean@lri.fr

Abstract

In this article, we provide a Coq mechanised, executable, formal semantics for a realistic fragment of SQL consisting of `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* sub-queries. Relying on the Coq extraction mechanism to Ocaml, we further produce a Coq certified semantic analyser for a SQL compiler. We then relate this fragment to a Coq formalised (extended) relational algebra that enjoys a *bag* semantics hence recovering all well-known algebraic equivalences upon which are based most of compilation optimisations. By doing so, we provide the first *formally mechanised proof* of the *equivalence* of SQL and extended relational algebra.

CCS Concepts • **Information systems → Relational database query languages**; • **Theory of computation → Program semantics**;

Keywords SQL, Formal Semantics, Coq

ACM Reference Format:
Véronique Benzaken and Évelyne Contejean. 2018. A Coq Mechanised Formal Semantics for Realistic SQL Queries: Formally Reconciling SQL and Bag Relational Algebra. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Providing a formal semantics for a language is a tricky but crucial task as it allows compilers to rigorously reason about program behaviours and to verify the correctness of optimisations [13, 18]. When considering real-life programming languages the task becomes even harder as it happens that the specifications of the language are often written in natural language. Even when they are formal, they only account for a limited subset of the considered language and are, most of the time, human-checked proven correct. In all cases, there are few strong guarantees that the whole faithfully accounts for the exact semantics and correctness of performed optimisations. As *the* standard for relational database management systems (RDBMS's), SQL is widely and intensively used and does not escape these drawbacks.

To obtain such high level guarantees, a promising approach consists in using *proof assistants* such as Coq [20] or Isabelle [21] to define *mechanised, executable* semantics whose correctness is *machine-checkable*. A shining demonstration of the viability of this undertaking for real systems is Leroy's CompCert project [15].

Our long-term goal, based on the same methodology, aims at providing a Coq verified compiler for SQL. In this article we focus on semantic issues and define, using Coq, a formal semantics for SQL.

SQL's Specificities Over time, the search for such a formal semantics has been pursued by the database community. However only restricted fragments of the language were handled. This is due, in particular, to SQL's specificities.

SQL's main construct consists in the `select from where group by having` block. It is well known that SQL's `select from where` enjoys a *bag* semantics, the same element may occur several times in the result, while purely set-theoretic operators such as ∪ (`union`), ∩ (`intersect`) and \ (`except`) have a set-theoretic one. Any formal semantics must account for both sets and bags.

It is notorious that SQL deals with `NULL` *values* that are intended to represent unknown information. This aspect, considered of major difficulty in the database field, has to be accurately handled by any mechanised semantics.

SQL queries involve *functions* (`+`, `-`, ...) and *aggregates* (`sum`, `max`, `count`, ...) that are intensively used namely when the `group by having` clause is present.

Last, SQL allows for nested sub-queries, *i.e.*, queries gathering different `select`'s, and more importantly *correlated*

*Work funded by the DataCert ANR project: ANR-15-CE39-0009.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

sub-queries, *i.e.*, sub-queries with free variables as in `select a1 from t1 group by a1 having exists (select a2 from t2 group by a2 having sum(1+0*a1) = 0);` where `a1` is free in the innermost `select`.

Any accurate semantics for SQL must account for those *four features*. When considered *independently* those four aspects are merely technical to handle. As will be demonstrated in this work, when considered *all together* as we do, the situation gets really challenging due to SQL's peculiar environments' management. This explains why no formal semantics for SQL exists to date.

Related Works As relational algebra [8] plays for SQL the same role as (a combinator version of) λ-calculus for functional programming languages, obviously, early proposals, among those [5, 17], presented a translation from SQL to such an algebra (see [8] for a survey). Only a subset of SQL (with no functions, nested queries, `NULL`'s nor bags though) was addressed. The first semantics for SQL accounting for `NULL`'s and bags is found in [12]. However, the work does not consider `group by having` clauses, aggregates, quantifiers in formulae nor *complex* expressions: their expressions are restricted to attributes' names or constants. As will be shown in Section 2 and 3, the treatment of complex expressions is very subtle.

On the proof assistant side, the first attempt to formalise relational algebra, using the Agda proof assistant [19], is found in [9, 10] while the first, almost complete, Coq formalisation of the relational model in which the data model, algebra, tableaux queries, the chase as well as integrity constraints aspects were modelled, is found in [3].

The very first attempt to verify, using Coq, a RDBMS is presented in [16]. However the SQL fragment considered is a reconstruction of SQL in which attributes are denoted by positions. They did not deal with `group by having` clauses, quantifiers in formulae, nested, correlated queries neither `NULL`'s nor aggregates. As we shall see in Section 2 and 3 the treatment of attributes' names and more generally environments in the presence of *correlated sub-queries* is a particularly, if not the most, tough task. Recently, in [6], a tool to decide whether two SQL queries are equivalent is presented. To do so, the authors defined HottSQL, a K-relation [11] based semantics for SQL which handles the `select from where` fragment with aggregates but without `having`. Like [16], they used a reconstruction of the language thus not accounting for the trickier aspects of variable binding. Last, and more importantly, their semantics is *not executable* hence it is impossible to verify whether they do implement SQL's semantics. Finally, the closest proposal in terms of mechanised semantics for SQL is addressed in [2]. A mere translation from SQL to a Coq modelisation of the nested relational algebra (NRA [7]) which *directly* serves as a semantics for SQL is proposed. However, as NRA's syntax significantly deviates from SQL's one, convincing oneself of the accuracy of the proposed semantics is not immediate.

Contributions In this article we give five contributions. We provide *(i)* SQL_{Coq}, a Coq internalisation of SQL's syntax together with its mechanised *executable* bag semantics. Our formalisation accounts for `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* subqueries. By doing so, thanks to the Coq extraction mechanism to Ocaml, we produce *(ii)* a Coq certified semantic analyser for the compiler. We then formalise, using Coq, *(iii)* SQL_{Alg}, a *bag*, *environment-aware* relational algebra similar to, while extending it, the one presented in [8]. By formally relating SQL_{Coq} to SQL_{Alg}, through a Coq mechanised translation, and by formally proving that these translations preserve semantics, we not only are able to *(iv)* recover all well-known algebraic equivalences on which are based most of compilers' optimisations but we also *(v)* establish the first, to our best knowledge, *mechanised*, *formal* proof of *equivalence* between the considered SQL fragment and bag relational algebra.

Organisation In Section 2, we first present SQL and SQL's subtleties that need to be taken into account to provide a correct semantics. Then we detail, in Section 3, SQL_{Coq}'s syntax and semantics. Section 4 is devoted to the mechanisation of SQL_{Alg}, the bag algebra. Then the translations between SQL_{Coq} and SQL_{Alg} as well as the equivalence theorem are presented. We conclude, draw lessons and give perpectives in Section 5.

2 SQL: Simple and Complex

As a domain-specific declarative language SQL is often considered simple.

2.1 SQL in a Nutshell

SQL operates over collections of tuples. Tuples are labelled records whose components range over atomic types (integers, strings, etc). In this context labels are called *attributes*. Such attributes are denotable and may be used within expressions built thanks to functions and accumulators which are called *aggregates*. SQL's main construct consists in

`select e from q where c1 group by e' having c2.`

For instance

```
select a+2 as a', max(c) as mc from t where b>3
      group by a having sum(c) = 0
```

is a typical SQL query where `a,b,c` are attributes, `sum`, `max` are aggregates, `+` a function and `b>3` a condition.

In order to evaluate a block, inner query `q` is evaluated, then filtered against boolean condition `c1`. This intermediate result is so-called "grouped" using expressions `e'`; meaning that all elements of a group are homogeneously evaluated over `e'`, and groups are maximal w.r.t. inclusion. Then groups are filtered against boolean condition `c2`, and expressions

e are evaluated over remaining groups, yielding the final tuples. On the example above, assuming that the evaluation of t yields

$$\left\{ \begin{array}{l} (a=1;b=1;c=4),(a=1;b=5;c=2),(a=1;b=4;c=-2), \\ (a=3;b=5;c=1),(a=3;b=4;c=2) \end{array} \right\}$$

The filtering step against b>3 gives

$$\left\{ \begin{array}{l} (a=1;b=5;c=2),(a=1;b=4;c=-2), \\ (a=3;b=5;c=1),(a=3;b=4;c=2) \end{array} \right\}$$

The grouping step w.r.t. a provides

$$\left\{ \begin{array}{l} \{(a=1;b=5;c=2),(a=1;b=4;c=-2)\}, \\ \{(a=3;b=5;c=1),(a=3;b=4;c=2)\} \end{array} \right\}$$

The filtering step against sum(c)=0 discards the last group for which sum(c) = 3 and keeps only the first group. The evaluation of a+2, max(c) yields the result {(a'=3;mc=2)}.

2.2 Inside SQL

Yet, SQL's semantics is more subtle than appears at first sight. It is described by the ISO Standard [14] which is difficult to exploit as:

- it consists of thousand pages hence it is complex to find where features are described (most of the time a single piece of information is spread over the whole document)
- when described, features are depicted in a verbose way (for instance definition of the OCaml equivalent of map takes several pages)
- last, even abundant, it is often under-specified.

For all those reasons it cannot serve as a formal semantics. This also explains why many vendors implement various aspects of it in their own way as witnessed by [1]. Of course, we relied on the Standard _ as much as possible _ and meanwhile we tested our development against systems like PostgreSQL and Oracle™ to better grasp SQL's semantics.

In the remainder of this article, we note $[\![q]\!]$ the result of the evaluation of query q, () the tuple constructor, [ ] the list constructor, { } the set constructor and {| |} the bag constructor. Figure 1 gathers a bunch of queries that will illustrate SQL's most subtle aspects.

2.2.1 NULL Values

SQL deals with NULL values that are intended to represent unknown information. A 3-valued logic combined with the classical Boolean logic is used to handle them. However NULL's are not treated in a uniform way according to the context as we illustrate in this section.

The first query returns {|(b = 3);(b = NULL)|} thus exemplifying that NULL behaves as an absorbing element for functions[1]. The next three queries, borrowed from [12], exemplify the fact that NULL is neither equal to nor different from any other value (including itself): comparing NULL with any expression always yields unknown. Query Q2 returns an empty result. This is explained by the fact that

$$[\![\texttt{select s.a from s}]\!] = \{|(\texttt{s.a=NULL})|\}\,,$$

hence [[r.a not in select s.a from s]] (where in is the membership predicate) yields not unknown, that is unknown, over all tuples (r.a=x), in particular over (r.a=1) and (r.a=NULL). This condition is eventually considered as false, therefore neither tuple belongs to the result of query Q2.

Query Q3 returns {|(r.a=1);(r.a=NULL)|}. Let subQ3 be the sub-query (select * from s where s.a = r.a), it yields an empty result over all tuples (r.a=x), hence [[exists (subQ3)]], where exists stands for the non emptyness predicate, is always false and [[not exists (subQ3)]] is always true, thus (r.a=1) and (r.a=NULL) are in the result of Q3.

Query Q4 returns {(r.a=1)}, because the set difference does not use 3-valued logical equality, but standard syntactic equality. Here both tuples (r.a=NULL) and (s.a=NULL) are equal. The main concern is to precisely detect when SQL falls from 3-valued logic into usual Boolean logic. It appears that whereas the evaluation of formulae is performed in the 3-valued logic, when using them as filtering conditions (either where or having) usual Boolean logic is used (unknown becomes false). This will be detailed in Figure 7 and Figure 8 of Section 3.

Last, query Q5 returns {|(t.a=NULL,c=2);(t.a=1,c=1)|}. This illustrates the fact that NULL, which is neither equal nor different from NULL in a 3-valued logic, is indeed equal to NULL in the context of grouping. The semantics proposed in Section 3.2 will account for such behaviours.

2.2.2 Correlated Sub-queries

Let us now address the way SQL manages evaluation environments in presence of aggregates and nested *correlated queries*. In order to evaluate simple (without aggregates) expressions, it is enough to have a single environment, containing information about the bound attributes and the values for them. In this simple case (*e.g.*, select a1, b1 from t1;) such an environment corresponds to a unique tuple (a1=x,b1=y) where x and y range in the active domains of a1 and b1 respectively.

Evaluating expressions with aggregates is more involved, since an aggregate operates over a list of values, each one corresponding to a tuple. The crucial point is to understand how such a list of tuples, that we call an *evaluation context*, is produced. Section 10.9 of [14] (< aggregate functions >, *how to retrieve the rows* – page 545) should provide some guidance in answering this question. Unfortunately it was of no help. Taking advantage of Coq's execution mechanism we rather decided to run queries over a small database of our own. We thus proceeded by testing many queries over PostgreSQL, Oracle™ and against our formal semantics. Relying on a *mechanised and executable* semantics was essential as

[1]Except for Boolean functions in PostgreSQL.

r
a
1
NULL

s
a
NULL

t
a
1
NULL
NULL

t_1

a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1
1	1	1	6	2	1	2	6	3	1	4	6
1	2	1	7	2	2	2	7	3	2	4	7
1	3	1	8	2	3	2	8	3	3	4	8
1	4	1	9	2	4	2	9	3	4	4	9
1	5	1	10	2	5	2	10	3	5	4	10

t_2	
a_2	b_2
7	7
7	7

```
-- Q1
select r.a+2 as b from r;
-- Q2
select r.a from r where r.a not in (select s.a from s);
-- Q3
select r.a from r where not exists
  (select * from s where s.a = r.a);
-- Q4
select r.a from r except select s.a from s;
-- Q5
select t.a,count(*) as c from t group by t.a;
-- Q6
select a1, max(b1) as m from t1 group by a1;
-- Q7(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*a2) = k);
-- Q8(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*a1) = k);
```

```
-- Q9(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1) = k);
-- Q10(k), k =12
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*a1)+sum(1+0*a2) = k);
-- Q11(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*a1+0*a2) = k);
-- Q12(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*a1+0*b2) = k);
-- Q13(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*b1+0*b2) = k);
-- Q14(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1+0*b1+0*a2) = k);
```

Figure 1. Semantically Subtle Queries.

it compelled us to examine in details all sub-cases. It took us significant effort to reach the *semantically relevant* set of queries which are given in Figure 1. For all of them, we obtained the same results on all three systems. Let us comment on these queries. For `Q6` the result is:

$$\{\!|\ (\texttt{a1=1,m=10});(\texttt{a1=2,m=10});(\texttt{a1=3,m=5});(\texttt{a1=4,m=10})\ |\!\}$$

It is easy to understand what happens when evaluating `max(b1)` in `Q6`: each group (where `a1` is fixed) contains some tuples, each of them yielding a value for `b1`. Then `max` is computed over this list of values. For instance, the group T_1 where `a1=1` contains exactly one occurence of tuples of the form `(a1=1,b1=i)`, where `i` ranges from 1 to 10, hence `b1` ranges from 1 to 10, and `max(b1)` is equal to 10, whereas the group where `a1=3` contains tuples `(a1=3,b1=i)`, where `i=1,...,5`, and `max(b1)` is equal to 5. In this simple case a group merely yields an evaluation context – we say that the group has been *split into* individual tuples.

The situation gets more complex when evaluating an aggregate expression in a nested sub-query. How to build, in that case, the suitable evaluation contexts in order to get the needed lists of values, arguments of the aggregate? The last queries of Figure 1: `Q7,Q8,Q9,Q10,Q11,Q12,Q13,Q14` and table `t2` have been designed to answer this question. Note that all are *correlated* queries except `Q7` and `Q9` which are simply nested. These queries follow the same pattern:

```
select a1 from t1 group by a1 having
  exists (select a2 from t2 group by a2 having e=k);
```

Assuming that an aggregate expression occurs in a sub-query under more than two grouping levels, as `e` in the above pattern, there are several groups in the *environment*. In our example, homogeneous groups w.r.t. `a1` are the following:

$$\mathscr{G}_1 = \left\{ \begin{array}{ll} \bigcup_{i=1}^{10} \{\!|(\texttt{a1=1;b1=i})|\!\} & \bigcup_{i=1}^{10} \{\!|(\texttt{a1=2;b1=i})|\!\} \\ \bigcup_{i=1}^{5} \{\!|(\texttt{a1=3;b1=i})|\!\} & \bigcup_{i=6}^{10} \{\!|(\texttt{a1=4;b1=i})|\!\} \end{array} \right\}$$

whereas there is a single homogeneous group w.r.t. `a2`, $T_2 = \{\!|(\texttt{a2=7;b2=7});(\texttt{a2=7;b2=7})|\!\}$. Thus global environment for the innermost expression `e` is made of T_2 and one T_1 of $\mathscr{G}_1$ denoted by: $[T_2; T_1]$.

How to combine these groups in order to obtain the correct evaluation context (*i.e.*, the one actually used by SQL on PostgreSQL and Oracle™)? Which groups must be split and which must not?

Let us consider `Q7(k)` where the innermost filtering condition is `sum(1+0*a2) = k`. When $k \neq 2$ $[\![\texttt{Q7(k)}]\!]$ is empty, and for $k = 2$ it is equal to $\bigcup_{i=1}^{i=4} \{\!|(\texttt{a1=i})|\!\}$. Expression `sum(1+0*a2)` actually computes the number of tuples in the evaluation context and this number is equal to 2 for all groups T_1 of $\mathscr{G}_1$: combining T_1 and T_2 yields a context containing two (the cardinality of T_2) tuples, whatever the cardinality of T_1. We have to draw the conclusion that when

evaluating `sum(1+0*a2)` = k, T_2 has been split whereas T_1 is not used at all.

Let us now examine *correlated* query `Q8(k)`, which is very similar to `Q7(k)`, except that the filtering condition is `sum(1+0*a1)` = k. When $k \notin \{5, 10\}$ $[\![$`Q8(k)`$]\!]$ is empty while $[\![$`Q8(5)`$]\!]$ = $\{|$`(a1=3);(a1=4)`$|\}$ and $[\![$`Q8(10)`$]\!]$ = $\{|$`(a1=1);(a1=2)`$|\}$. Hence expression `sum(1+0*a1)` computes the cardinality of T_1. When evaluating `sum(1+0*a1)` = k, T_1 has been split, whereas T_2 is not used.

Fact 1. *In the same environment, $[T_2; T_1]$, SQL either splits T_1 or T_2 in order to build an evaluation context from which evaluation of aggregate expressions is performed. The expression to be evaluated directs the way the evaluation context is built by choosing which relevant group to split into: T_1 for* `1+0*a1`*, and T_2 for* `1+0*a2`*. At that point it seems that this choice is guided by attributes (*`a1` *or* `a2`*).*

Thus, the next interesting case is when there are no attributes in the expression under the aggregate as in `Q9(k)`. What should be the relevant group to be split into? Is there even such a relevant group for `sum(1)`? Actually `Q9(k)` yields the same result as `Q7(k)`, meaning that the relevant group for a constant is the innermost group T_2.

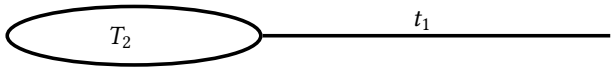
Fact 2. *In the very same environment,* `1+0*a2` *is equal to* `1` *and* `1+0*a1` *is not since these expressions are computed in different evaluation contexts: under aggregates, in SQL, usual arithmetic equalities are no longer valid.*

At that point, what happens if both expressions `1+0*a1` and `1+0*a2` have to be evaluated in the same environment as it is the case for `Q10(k)`, where `1+0*a1` and `1+0*a2` occur under *distinct* aggregates? There is no single obvious relevant group anymore. When $k \notin \{7, 12\}$ $[\![$`Q10(k)`$]\!]$ is empty while $[\![$`Q10(7)`$]\!]$ = $\{|$`(a1=3);(a1=4)`$|\}$ and $[\![$`Q10(12)`$]\!]$ = $\{|$`(a1=1);(a1=2)`$|\}$, meaning that both expressions `1+0*a1` and `1+0*a2` have been evaluated *independently*, the first in a context where T_1 has been split into, and the second where the split group is T_2.

Fact 3. *This makes clear that SQL allows two sub-expressions of a given expression to be evaluated in different contexts which is definitely contrary to what is done in other mainstream programming languages!*

What if `1+0*a1` and `1+0*a2` occur under the *same* aggregate, as in `Q11(k)`? When k=2, $[\![$`Q11(2)`$]\!]$ is $\bigcup_{i=1}^{i=4} \{|$`(a1=i)`$|\}$, otherwise $[\![$`Q11(k)`$]\!]$ is empty.

Fact 4. *Therefore T_2, the innermost relevant group, has been split into. T_1 has been collapsed to any of its elements t_1 since only its homogeneous part,* `a1`*, is used by the evaluation.*



As the reader may have noticed, all expressions under the aggregates were built upon grouping attributes. What happens when such is not the case? Query `Q12(k)` contains `sum(1+0*a1+0*b2)` and behaves exactly the same as `Q11(k)` does. Query `Q13(k)` contains `sum(1+0*b1+0*b2)` and is not *well formed* according to the Standard, thus, is not evaluated. The reason is that under the aggregate there are two non grouping attributes coming from different nesting levels. The last query, `Q14(k)`, which contains `sum(1+0*b1+0*a2)`, is also ill-formed and not evaluated. However one could imagine that it could have been accepted and evaluated in the following context:

At that point, we are able to sum up the lessons above and precisely explain how SQL manages environments.

2.3 Summary

First, when evaluating an expression with aggregates where the top operator is a function (for instance +, as in `Q10(k)`), each argument is evaluated separately.

Second, when evaluating an expression `ag`(e) where the top operator `ag` is an aggregate, this aggregate is evaluated over a list of values. The subtle point is to understand how to build the suitable evaluation context to retrieve such values. Let us clarify the previously discussed notions.

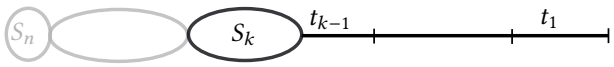
Complete Environments A complete environment, $\mathscr{E} = [S_n; \ldots; S_1]$, is a stack of *slices*: one slice per *nesting level* i, the innermost level being on the top. When necessary, we shall equally adopt the following notation for environments $\mathscr{E} = (A, G, T) :: \mathscr{E}'$ in order to highlight the list's head. Slices are of the form $S = (A, G, T)$, where A (also noted $A(S)$) contains the relevant attributes for that level of nesting, *i.e.*, the names introduced in the subquery at this level[2]; G the grouping expressions appearing in the `group by` (also noted $G(S)$); and T a non empty list of tuples[3] (also noted $T(S)$).

Evaluation Contexts When e is a constant expression, the list of tuples $T(S_n)$ comes from the innermost slice of environment $\mathscr{E} = [S_n; \ldots; S_1]$. In the simple case where all attributes of e are introduced at the same level i, the relevant list is simply $T(S_i)$. Otherwise, when attributes of e belong to at least two different levels, the innermost (*i.e.*, of greatest index) being S_k, there are two cases:

[2]If this subquery is a `select from ...` these are the names in the `select`.

[3]When there is a grouping clause at this level, it is an homogeneous group, otherwise it is a single tuple.

- either the expression is not well-formed (*cf* `Q13` and `Q14`), because e contains an expression of $T(S_j), j < k$ which is not grouped.
- or the expression e is exactly built upon the attributes corresponding to the kth level and the grouping expressions[4] of outermost levels $k-1, \ldots, 1$. In this case, let t_j be a fixed tuple chosen in each $T(S_j)$ for $j < k$, then the list of relevant tuples is made of the concatenations $(t; t_{k-1}; \ldots; t_1)$, where t ranges over $T(S_k)$.



We are now able to present our Coq mechanised formal semantics.

3 A Formal Coq Mechanised Semantics for SQL

SQL_{Coq} addresses the fragment consisting of `select [distinct] from where group by having` queries with `NULL` *values, functions, aggregates, quantifiers* and *nested* potentially *correlated* (in `from`, `where` and `having` clauses) subqueries. It accounts for `in`, `any`, `all` and `exists` constructs and assigns queries a Coq mechanised (bag) semantics that complies with the Standard.

3.1 SQL_{Coq}: Syntax

SQL_{Coq}'s syntax is given on Figure 2, Figure 3 and Figure 4 where the left part of figures represents SQL_{Coq}'s abstract syntax and the right part the corresponding Coq syntax. We assume that we are given attributes, functions and aggregates. We shall allow strings, integers and booleans to be values, as well as the special `NULL`. On the top of them, we define usual expressions, first without aggregates e^f, and then with aggregates e^a. SQL formulae are similar to first-order formulae except they are always interpreted in a finite domain, which is syntactically refered to as `dom` in Figure 3. Such formulae will then be used in the context of SQL_{Alg}.

SQL_{Coq} sticks, syntactically, as much as possible, to SQL's syntax but the SQL-aware reader shall notice that SQL_{Coq} slightly 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`. Table below, summarises the way the parser handles the different cases for translating the absence or presence of `group by` into an explicit grouping construct in SQL_{Coq}.

SQL			SQL_{Coq}
aggregate (in select)	group by	having	
?	g	?	Group_By g
✓	✗	?	Group_By nil
?	✗	✓	Group_By nil
✗	✗	✗	Group_Fine

`Group_Fine` corresponds to the finest partition[5] and differs from `Group_By` $[a_1, \ldots, a_n]$ where $[a_1, \ldots, a_n]$ is the list of labels of the current query. The construct `Group_By` $[a_1, \ldots, a_n]$ is used in SQL_{Coq} to encode the `distinct` construct of SQL. `Group_By nil` corresponds to the coarse partition. We also force explicit and mandatory renaming of attributes, when $*$ is not used. In our syntax, `select a, b from t;` is expressed by

```
select a as a, b as b from (table t[*])
    where true group by Group_Fine 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 functions ($\mathfrak{fn}$) and are evaluated on *tuples* while the latter also allow unnested[6] aggregates ($\mathfrak{ag}$) and are, in that case, evaluated on *collections of tuples*. In the same line, we used the same language 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. This last case is handled by renaming all query's attributes as expressed on last line of Figure 4.

3.2 SQL_{Coq}: Semantics

Given a tuple t we note $\ell(t)$ the attributes occuring in t. We assume that we are given a database instance $[\![\_]\!]_{db}$ defined as a function from relation names to *bags* of tuples[7] as well as predefined, fixed interpretations, $[\![\_]\!]_p$, for predicates $\mathfrak{pr}$[8], *i.e.*, a function from vectors of values to Booleans, $[\![\_]\!]_a$ and $[\![\_]\!]_f$ for aggregates $\mathfrak{ag}$ and functions $\mathfrak{fn}$ respectively[9]. As established in Section 2, (complex) expressions occuring in (possibly correlated sub-) queries, are evaluated under a sliced environment, $\mathscr{E} = [S_n; \ldots; S_1]$ (or $\mathscr{E} = (A, G, T) :: \mathscr{E}'$), the innermost level, n, corresponding to the first slice. The evaluation of a syntactic entity e of type x in environment $\mathscr{E}$ will be denoted by $[\![e]\!]^{\text{x}}_{\mathscr{E}}$ (where x is f for expressions built only with functions, a for expressions built also with aggregates, b for formulae and q for queries).

[4] Those appearing in the `group by` clause of the level ; when there are no such grouping expressions, all attributes of the level are allowed.

[5] The partition consisting of the collection of singletons, one singleton for each tuple.

[6] 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)`

[7] These multisets enjoy some list-like operators such as `empty`, `map`, `filter`, etc.

[8] $\mathfrak{pr}$ is $<$, `in` etc.

[9] a may be `sum`, `count` etc. and f: `+`, `*`, `-` etc.

function ::= + | − | ∗ | / | ... | *user defined fun*
aggregate ::= sum | avg | min | ... | *user defined ag*
value ::= string val | integer val | bool val | NULL
e^f ::= value | attribute | function($\overline{e^f}$)
e^a ::= e^f | aggregate(e^f) | function($\overline{e^a}$)

```
Inductive value : Set :=
  | String : string → value
  | Integer : Z → value
  | Bool : bool → value
  | NULL : value.
Inductive funterm : Type :=
   | F_Constant : value → funterm
   | F_Dot : attribute → funterm
   | F_Expr : symb → list funterm → funterm.
Inductive aggterm : Type :=
   | A_Expr : funterm → aggterm
   | A_agg : aggregate → funterm → aggterm
   | A_fun : symb → list aggterm → aggterm.
```

Figure 2. Expressions.

formula ::=
| formula (**and** | **or**) formula
| **not** formula
| **true**
| $p(\overline{e^a})$ $p \in$ *predicate*
| $p(\overline{e^a}, (\textbf{all} \mid \textbf{any})$ dom) $p \in$ *predicate*
| $\overline{e^a}$ **as** attribute **in** dom
| **exists** dom

```
Inductive conjunct : Type := And | Or.
Inductive quantifier : Type := All | Any.

Inductive select : Type := Select_As : aggterm → attribute → select.

Inductive formula (dom : Type): Type :=
  | Conj : conjunct → formula dom → formula dom → formula dom
  | Not : formula dom → formula dom
  | True : formula dom
  | Pred : predicate → list aggterm → formula dom
  | Quant : list aggterm → predicate → quantifier → dom → formula dom
  | In : list select → dom → formula dom
  | Exists : dom → formula dom.
```

Figure 3. Formulae, Parameterized by a Finite Domain of Interpretation dom.

select_item ::= ∗ | $\overline{e^a \textbf{ as attribute}}$
query ::=
| table
| query (**union** | **intersect** | **except**) query
| **select** select_item
from $\overline{\text{from\_item}}$
(**where** formula)
(**group by** $\overline{e^f}$ (**having** formula))
from_item ::= query($\overline{\text{attribute as attribute}}$)

```
Inductive select_item : Type :=
  Select_Star | Select_List : list select → select_item.

Inductive att_renaming : Type :=
   Att_As : attribute → attribute → att_renaming.
Inductive att_renaming_item : Type :=
   Att_Ren_Star | Att_Ren_List : list att_renaming → att_renaming_item.

Inductive group_by : Type :=
   Group_Fine | Group_By : list funterm → group_by.

Inductive set_op : Type := Union | Intersect | Except.

Inductive query : Type :=
  | Table : relname → query
  | Set : set_op → query → query → query
  | Select : (** select *) select_item  →
             (** from *) list from_item →
             (** where *) formula query →
             (** group by *) group_by →
             (** having *) formula query → query

with from_item : Type :=
  From_Item : query → att_renaming_item → sql_from_item.
```

Figure 4. SQL_{Coq} Syntax

The semantics of simple expressions, which poses no difficulties, is given in Figure 5. The semantics of complex expressions detailed in Figure 6, deserves comments. When the complex expression is headed by a function, $\mathfrak{f}\mathfrak{n}(\overline{e})$, it simply amounts to a recursive call. When the complex expression is of the form $\mathfrak{ag}(e)$, according to Section 2, one has first to find the suitable level of nesting for getting the group to be split into. Then, produce the list of values by evaluating e, and then compute the evaluation of $\mathfrak{ag}$ against this list of values. In environment $\mathscr{E}$=$[S_n; \ldots S_1]$, level i is a suitable candidate expressed by $\mathbb{S}_e(A(S_i), [S_{i-1}; \ldots; S_1], e)$ on Figure 6 whenever e is built upon $G = A(S_i) \cup \bigcup_{j<i} G(S_j)$ which is in turn expressed by $\mathbb{B}_u(G, e)$ on Figure 6. When e is a constant, the

$$
\begin{array}{lcll}
[\![c]\!]^{\mathrm{f}}_{\mathscr{E}} & = & \mathrm{c} & \text{if } c \text{ is a value} \\
[\![a]\!]^{\mathrm{f}}_{[]} & = & \textbf{default} & \text{if } a \text{ is an attribute} \\
[\![a]\!]^{\mathrm{f}}_{(A,G,[])::\mathscr{E}} & = & [\![a]\!]^{\mathrm{f}}_{\mathscr{E}} & \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathscr{E}} & = & t.\mathrm{a} & \text{if } a \in \ell(t) \\
[\![a]\!]^{\mathrm{f}}_{(A,G,t::T)::\mathscr{E}} & = & [\![a]\!]^{\mathrm{f}}_{\mathscr{E}} & \text{if } a \notin \ell(t) \\
[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{f}}_{\mathscr{E}} & = & [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{f}}_{\mathscr{E}}}) & \\
 & & & \text{if } \mathfrak{fn} \text{ is a function,} \\
 & & & \text{and } \overline{e} \text{ is a list of simple expressions}
\end{array}
$$

```
(* The type of evaluation environnements *)
Definition env_type := list (list attribute * group_by * list tuple).

Fixpoint interp_dot env (a : attribute) :=
  match env with
    | nil ⇒ default_value a
    | (sa, gb, nil) :: env' ⇒ interp_dot env' a
    | (sa, gb, t :: l) :: env' ⇒
       if a inS? labels t then (dot t a) else interp_dot env' a
   end.
Fixpoint interp_funterm  env t :=
  match t with
    | F_Constant c ⇒ c
    | F_Dot a ⇒ interp_dot env a
    | F_Expr f l ⇒
       interp_symb f (List.map (fun x ⇒ interp_funterm env x) l)
  end.
```

Figure 5. Simple Expressions' Semantics.

$$
\frac{c \in \mathscr{V}}{\mathbb{B}_u(G,c)} \qquad \frac{e \in G}{\mathbb{B}_u(G,e)} \qquad \frac{\bigwedge_{\overline{e}} \mathbb{B}_u(G,e)}{\mathbb{B}_u(G,\mathfrak{fn}(\overline{e}))}
$$

$$
\frac{\mathbb{B}_u((A \cup \bigcup_{(A',G,T)\in\mathscr{E}} G), e)}{\mathbb{S}_e(A,\mathscr{E},e)}
$$

$$
\frac{c \in \mathscr{V}}{\mathbb{F}_e(\mathscr{E},c) = \mathscr{E}} \qquad \frac{e \notin \mathscr{V}}{\mathbb{F}_e([],e) = \mathsf{undefined}}
$$

$$
\frac{e \notin \mathscr{V} \quad \mathbb{F}_e(\mathscr{E},e) = \mathscr{E}'}{\mathbb{F}_e(((A,G,T)::\mathscr{E}),e) = \mathscr{E}'}
$$

$$
\frac{\mathbb{F}_e(\mathscr{E},e) = \mathsf{undefined} \quad \mathbb{S}_e(A,\mathscr{E},e)}{\mathbb{F}_e(((A,G,T)::\mathscr{E}),e) = (A,G,T)::\mathscr{E}}
$$

$$
[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{a}}_{\mathscr{E}}})
$$

$$
[\![\mathfrak{ag}(e)]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{ag}]\!]_a\left(\overline{[\![e]\!]^{\mathrm{f}}_{((A,G,[t])::\mathscr{E}')}}\right)_{t\in T}
\quad \text{iff } \mathbb{F}_e(\mathscr{E},e) = (A,G,T)::\mathscr{E}'
$$

```
Fixpoint (* (𝔹_u(G, f)) *) is_built_upon G f :=
  match f with
    | F_Constant _ ⇒ true
    | F_Dot _ ⇒ f inS? G
    | F_Expr s l ⇒ (f ins? G) || forallb (is_built_upon G) l
  end.

Definition (* (𝕊_e(la, env, f)) *) is_a_suitable_env la env f :=
  is_built_upon
     (map (fun a ⇒ F_Dot a) la ++
     flat_map (fun slc ⇒ match slc with (_, G, _) ⇒ G end) env)
     f.

Fixpoint (* (𝔽_e(env, f)) *) find_eval_env env f :=
  match env with
    | nil ⇒ if is_built_upon nil f then Some nil else None
    | (la1, g1, l1) :: env' ⇒
      match find_eval_env env' f with
        | Some _ as e ⇒ e
        | None ⇒ if is_a_suitable_env la1 env' f then Some env else None
      end
  end.

Fixpoint interp_aggterm env (ag : aggterm) :=
  match ag with
  | A_Expr ft ⇒
       (* simple expression without aggregate *) interp_funterm env ft
  | A_fun f lag ⇒
       (* simple recursive call in order to evaluate independently
          the sub-expressions when the top symbol is a function *)
      interp_symb f (List.map (fun x ⇒ interp_aggterm env x) lag)
  | A_agg ag ft ⇒
    let env' :=
       if is_empty (att_of_funterm ft)
       then (* the expression under the aggregate is a constant *) Some env
       else (* find the outermost suitable level *) find_eval_env env ft in
    let lenv :=
        match env' with
          | None | Some nil ⇒ nil
          | Some ((la1, g1, l1) :: env'') ⇒
              (* the outermost group is split into *)
               map (fun t1 ⇒ (la1, g1, t1 :: nil) :: env'') l1
        end in
        interp_aggregate ag (List.map (fun e ⇒ interp_funterm e ft) lenv)
  end.
```

Figure 6. Complex (with Aggregates) Expressions' Semantics.

innermost level is chosen (here n), otherwise, the outermost suitable candidate level is chosen as expressed by $\mathbb{F}_e(\mathscr{E},e)$.

Formulae's semantics, given in Figure 7, relies on expressions' semantics. As the syntax is parametrised by a domain dom, similarly formulae's semantics is parametrised by the domain's evaluation. This is expressed, in the Coq development, by `Hypothesis I : env_type → dom → bagT.`, and is

$$
\begin{array}{lcl}
[\![f_1 \textbf{ and } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \wedge [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![f_1 \textbf{ or } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \vee [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{not } f]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \neg [\![f]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{true}]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
[\![\mathrm{pr}(\overline{e_i})]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![\mathrm{pr}]\!]_p(\overline{[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \\
[\![\mathrm{pr}(\overline{e_i}, \textbf{ all } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{r}{\text{iff } [\![\mathrm{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ for all } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\mathrm{pr}(\overline{e_i}, \textbf{ any } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{r}{\text{iff } [\![\mathrm{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ for at least one } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\overline{e_i \textbf{ as } a_i} \textbf{ in } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{r}{\text{if } (\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \text{ belongs to}^{\dagger} [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\textbf{exists } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \text{ iff } [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \text{ is not empty}
\end{array}
$$

†See paragraph for NULL's in Section 3.2.

```
Hypothesis I : env_type → dom → bagT.
Fixpoint eval_formula env (f : formula) : Bool.b B :=
  match f with
  | Sql_Conj a f1 f2 ⇒
      (interp_conj B a) (eval_formula env f1) (eval_formula env f2)
  | Sql_Not f ⇒ Bool.negb B (eval_formula env f)
  | Sql_True ⇒ Bool.true B
  | Sql_Pred p l ⇒ interp_predicate p (map (interp_aggterm env) l)
  | Sql_Quant qtf p l sq ⇒
    let lt := map (interp_aggterm env) l in
    interp_quant B qtf
                 (fun x ⇒ let la := Fset.elements _ (labels x) in
                    interp_predicate p (lt ++ map (dot x) la))
                 (Febag.elements _ (I env sq))
  | Sql_In s sq ⇒
      let p := (projection env (Select_List s)) in
      interp_quant B Exists_F
        (fun x ⇒ match Oeset.compare OTuple p x with
                  | Eq ⇒ if contains_null p
                            then unknown else Bool.true B
                  | _ ⇒ if (contains_null p || contains_null x)
                            then unknown else Bool.false B
                  end)
        (Febag.elements _ (I env sq))
  | Sql_Exists sq ⇒
       if Febag.is_empty _ (I env sq) then Bool.false B else Bool.true B
  end.
```

Figure 7. Formulae's Semantics.

$$
\begin{array}{lcl}
[\![tbl]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![tbl]\!]_{db} \qquad \text{if } tbl \text{ is a table} \\
[\![q_1 \textbf{ union } q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \textbf{ intersect } q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \textbf{ except } q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
\multicolumn{3}{l}{[\![\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w} \\
\multicolumn{3}{r}{\textbf{group by } G \textbf{ having } h]\!]^{\mathrm{q}}_{\mathscr{E}} =} \\
\multicolumn{3}{r}{\left\{\overline{\left(a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(T),G,T)::\mathscr{E}}\right)} \,\middle|\, T \in \mathbb{F}_3\right\}} \\
\multicolumn{3}{l}{\text{if } F = \bowtie_i [\![f_i]\!]^{\mathrm{from}}_{\mathscr{E}}} \\
\multicolumn{3}{l}{\text{and } F_1 = \left\{t \in F \,\middle|\, [\![w]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top\right\}} \\
\multicolumn{3}{l}{\text{and } \mathbb{F}_2 \text{ is a partition}^{\dagger} \text{ of } F_1 \text{ according to } G} \\
\multicolumn{3}{l}{\text{and } \mathbb{F}_3 = \left\{T \in \mathbb{F}_2 \,\middle|\, [\![h]\!]^{\mathrm{b}}_{(\ell(T),G,T)::\mathscr{E}} = \top\right\}} \\
[\![\overline{q(a_i \textbf{ as } b_i)}]\!]^{\mathrm{from}}_{\mathscr{E}} & = & \{\overline{(b_i = c_i)} \mid \overline{(a_i = c_i)} \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}\}
\end{array}
$$

†See paragraph for NULL's in Section 3.2.

```
Fixpoint eval_sql_query env (sq : sql_query) {struct sq} :=
match sq with
 | Sql_Table tbl ⇒ instance tbl
 | Sql_Set o sq1 sq2 ⇒
   if sql_sort sq1 =S?= sql_sort sq2
   then Febag.interp_set_op _ o
          (eval_sql_query env sq1) (eval_sql_query env sq2)
   else Febag.empty _
 | Sql_Select s lsq f1 gby f2  ⇒
  let elsq :=
  (** evaluation of the from part *)
        List.map (eval_sql_from_item env) lsq in
  let cc :=
  (** selection of the from part by formula f1, with old names *)
     Febag.filter _
       (fun t ⇒
         Bool.is_true _
           (eval_sql_formula eval_sql_query (env_t env t) f1))
            (N_product_bag elsq) in
  (** computation of the groups grouped according to gby *)
  let lg1 := make_groups env cc gby in
  (** discarding groups according the having clause f2 *)
  let lg2 :=
    List.filter
      (fun g  ⇒
         Bool.is_true _
           (eval_sql_formula eval_sql_query (env_g env gby g) f2))
      lg1 in
  (** applying outermost projection and renaming, the select part s *)
   Febag.mk_bag BTupleT
     (List.map (fun g ⇒ projection (env_g env gby g) s) lg2)
  end

(** evaluation of the from part *)
with eval_sql_from_item env x :=
  match x with
  | From_Item sqj sj ⇒
    Febag.map BTupleT BTupleT
    (fun t ⇒
       projection (env_t env t) (att_renaming_item_to_from_item sj))
    (eval_sql_query env sqj)
    end.
```

Figure 8. SQL Queries' Semantics.

expanded as query interpretation, $[\![\_]\!]^{\mathrm{q}}_{\_}$, in the formal definition.

Let's finally comment on query semantics, $[\![\_]\!]^{\mathrm{q}}_{\_}$, given in Figure 8. For the set theoretic operators, we chose to assign them a bag semantics even if our notations do not explicitely mention `all`. If one wants to recover the usual set semantics for `sq = q1 op q2`, one has to apply duplicate elimination thanks to δ(`sq`) = `select * from` $\overline{\mathtt{sq}(a_i \text{ as } a_i)_{a_i \in \ell(sq)}}$ `group by` ℓ(`sq`). The most complex case is the `select from where group by having` one. Informally, a first step consists in evaluating the `from` and then filtering it thanks to the `where` formula.

More precisely how to check that a tuple t fullfils `where` condition w in context $\mathscr{E}$? According to the definition in Figure 7, w is evaluted *w.r.t* a single environment. This means that t and $\mathscr{E}$ have to be combined into this single environment, $\mathscr{E}'$ such that $[\![w]\!]^{\mathrm{f}}_{\mathscr{E}'}$ is equal to the evaluation of w, where the attributes a in $\ell(t)$ are bound to $t.a$, and the attributes a in $\bigcup_{S\in\mathscr{E}} A(S)$ are bounded thanks to $\bigcup_{S\in\mathscr{E}} A(T)$. This is exactly what is done when $\mathscr{E}' = (\ell(t), [], [t]) :: \mathscr{E}$.

Then the (intermediate) collection of tuples obtained is partitioned according to the grouping expressions in the `group by` G, yielding a collection of collections of tuples: the groups. When there is no grouping clause, the finest partition denoted `Group_Fine` in the Coq development is used.

The way groups are further filtered *w.r.t* the `having` condition h follows the same pattern as `where`, except that some complex expressions may occur in h. When evaluting an expression of the form $\mathfrak{ag}(e)$ for a group T, all tuples of the group are needed; when evaluting a simple expression, any tuple of T yields the same result, T being homogeneous *w.r.t* the grouping criterion G. Hence the proper evaluation environment for filtering the group T *w.r.t* h in environment $\mathscr{E}$ is $(\ell(T), G, T) :: \mathscr{E}$.

Last, the `select` clause is applied yielding again a collection of tuples as a result.

About `NULL`***'s*** At the expression level, `NULL`'s are simply handled by the fact that they behave as an absorbing element *w.r.t* functions and are simply discarded for aggregates except for `count(*)` where they contribute as 1. In our formalisation this is expressed as constraints over $[\![\_]\!]_a$ and $[\![\_]\!]_f$. For formulae, we used a 3-valued logic. The evaluation of $\mathfrak{pr}(\overline{e})$ in environment $\mathscr{E}$ is equal to unknown iff there exists e_i in $\overline{e}$ such that $[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}$ = `NULL`. As usual, unknown distributes according to well-known 3-valued logic rules. Quantifiers `all` and `any` are respectively seen as a finite conjunct and a finite disjunct in 3-valued logic. Last, $\overline{e \textbf{ as } a}$ **in** q is evaluated as a finite conjunct of $\bigwedge \overline{e = t.a}$ where t ranges in $[\![q]\!]^{\mathrm{q}}$, meaning that as soon as e or $t.a$ is evaluated to `null`, $\bigwedge \overline{e = t.a}$ is unknown. Eventually, when used into queries' evaluation, the evaluation of formulae yielding unknown results are cast into false. It should be noticed that even if `NULL` is not equal to nor different from `NULL` or any other value in the context of formulae, `NULL` is equal to `NULL` for grouping. This is taken into account in Figure 8 by a careful definition of partition and of `make_groups` in the Coq development.

4 $\mathrm{SQL}_{\mathrm{Alg}}$: A Coq Mechanised Algebra for SQL

In this section we relate $\mathrm{SQL}_{\mathrm{Coq}}$ to relational algebra in order to recover the well-known algebraic equivalences which are exploited by SQL compilers for optimisation purposes.

4.1 Relational Algebra in a Nutshell

The (extended) relational algebra, as presented in textbooks [8], consists of the well-known operators σ (selection), π (projection) and $\bowtie$ (join) completed with the γ (grouping) together with the set theoretic operators, intersection, union and difference. We focus on the former four operators.

$$q := r \mid \sigma_f(q) \mid \pi_S(q) \mid q \bowtie q \mid \gamma_{g,ag}(q)$$

In this setting, base relations, r are expressions. The selection operator, allows for filtering collections of tuples according whether they satisfy condition f. 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 [f].

The projection operator has the form π_W, and operates on all expressions, q, whose sort contains the subset of attributes 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 join operator, denoted $\bowtie$, takes arbitrary expressions q_1 and q_2 whose respective sorts are V and W, and allows to combine tuples from both operands. Its semantics is,

$$[\![q_1 \bowtie q_2]\!] = \{t \mid \exists v \in [\![q_1]\!], \exists w \in [\![q_2]\!], t|_V = v \wedge t|_W = w\}.$$

Quoting [8], "*operator $\gamma_{g,ag}$ partitions the tuples of q into groups. Each group consists of all tuples having one particular assignment of values to the grouping attributes in g. If there are no grouping attributes, the entire relation q is one group. For each group, one tuple consisting of the grouping attributes' values for that group and the aggregations, over all tuples of that group, for the aggregated attributes in ag is produced*". No further formal definition is given in [8]. The formal definition corresponding to it is given in Figure 11 where f is instanciated to true for this specific case.

4.2 $\mathrm{SQL}_{\mathrm{Alg}}$ Syntax and Semantics

As presented in [8], the algebra does not account for `having` conditions neither for complex expressions (grouping is only possible over attributes and aggregates are computed over single attributes) nor for environments. So as to deal with

SQL, ours is much expressive as it allows for grouping over simple expressions and allows complex expressions e^a in projections. In order to handle `having` conditions, that directly operate on groups, SQL_{Alg} extends what is presented in [8] by adding an extra parameter to γ: the `having` condition.

Expressions (simple and complex ones) as well as formulae[10] are shared with SQL_{Coq}. In order to define the semantics of SQL_{Alg}'s expressions, environments are needed, for the same reasons as for SQL_{Coq}: accounting for nesting. Hence SQL_{Alg} environments are the same as for SQL_{Coq}. What should be noticed is that $\bowtie$ is the true natural join, and that γ can be seen as a degenerated case of `select from where group by having`, where the `where` condition is absent (or set to `true`).

Based on our fornmalisation it is possible to define other, derived, operators such as the delta operator (intended to implement the `distinct`), the semi-join, anti-join and, provided that a `null` value be defined, the left and full outer-joins as illustrated on Figure 10. Let us at that point formally relate SQL_{Coq} and SQL_{Alg}.

4.3 SQL_{Coq} and SQL_{Alg} Are Equivalent

On Figure 12, we give $\mathbb{T}^{\text{q}}(\_)$ a translation from SQL_{Coq} to SQL_{Alg}, and its back translation $\mathbb{T}^{\text{Q}}(\_)$. Both use auxilliary translations ($\mathbb{T}^{\text{f}}(\_)$, resp. $\mathbb{T}^{\text{F}}(\_)$) which simply traverse formulae in order to translate the queries they contain. Since simple and complex expressions are shared, they are left unchanged by these translations. Notice that in order to translate the **Q_empty_tuple** construct from the algebra to a SQL query, one has to assume that the database schema contains at least a relation (**default_table**).

These translations are sound, provided that they are applied on "reasonable" database instances and queries.

Definition 4.1. A database instance $[\![\_]\!]_{db}$ is *well-sorted* if and only if all tuples in the same table have the same labels: $\forall r, t_1, t_2,\ t_1 \in [\![r]\!]_{db} \wedge t_2 \in [\![r]\!]_{db} \implies \ell(t_1) = \ell(t_2)$.

Definition 4.2. A SQL_{Coq} query sq is *well-formed* if and only if all labels in its `from` clauses are pairwise disjoint and its sub-queries are well-formed:

$$\frac{}{\mathbb{W}^{\text{q}}(tbl)}\ \text{if } tbl \text{ is a table}$$

$$\frac{\mathbb{W}^{\text{q}}(q_1)\quad \mathbb{W}^{\text{q}}(q_2)}{\mathbb{W}^{\text{q}}(q_1\ \textbf{union}\ q_2)} \qquad \frac{\mathbb{W}^{\text{q}}(q_1)\quad \mathbb{W}^{\text{q}}(q_2)}{\mathbb{W}^{\text{q}}(q_1\ \textbf{intersect}\ q_2)} \qquad \frac{\mathbb{W}^{\text{q}}(q_1)\quad \mathbb{W}^{\text{q}}(q_2)}{\mathbb{W}^{\text{q}}(q_1\ \textbf{except}\ q_2)}$$

$$\frac{\textbf{disjoint}\{\overline{b_i}\}_i \quad \bigwedge_i \mathbb{W}^{\text{q}}(q_i) \quad \mathbb{W}^{\text{f}}(w) \quad \mathbb{W}^{\text{f}}(h)}{\mathbb{W}^{\text{q}}(\textbf{select}\ s\ \textbf{from}\ \overline{q_i(a_i\ \textbf{as}\ b_i)}\ \textbf{where}\ w\ \textbf{group by}\ G\ \textbf{having}\ h)}$$

$$\frac{\mathbb{W}^{\text{f}}(f_1)\quad \mathbb{W}^{\text{f}}(f_2)}{\mathbb{W}^{\text{f}}(f_1\ \textbf{and}\ f_2)} \qquad \frac{\mathbb{W}^{\text{f}}(f_1)\quad \mathbb{W}^{\text{f}}(f_2)}{\mathbb{W}^{\text{f}}(f_1\ \textbf{or}\ f_2)} \qquad \frac{\mathbb{W}^{\text{f}}(f)}{\mathbb{W}^{\text{f}}(\textbf{not}\ f)}$$

$$\frac{}{\mathbb{W}^{\text{f}}(\textbf{true})} \qquad \frac{}{\mathbb{W}^{\text{f}}(\text{pr}(\overline{e}_i))} \qquad \frac{\mathbb{W}^{\text{q}}(q)}{\mathbb{W}^{\text{f}}(\textbf{exists}\ q)}$$

$$\frac{\mathbb{W}^{\text{q}}(q)}{\mathbb{W}^{\text{f}}(\text{pr}(\overline{e}_i, \textbf{all}\ q))} \qquad \frac{\mathbb{W}^{\text{q}}(q)}{\mathbb{W}^{\text{f}}(\text{pr}(\overline{e}_i, \textbf{any}\ q))} \qquad \frac{\mathbb{W}^{\text{q}}(q)}{\overline{e_i\ \textbf{as}\ a_i}\ \textbf{in}\ q}$$

Provided that those conditions be fullfilled we can state the following equivalence Theorem.

Theorem 4.3 ($\text{SQL}_{\text{Coq}} \equiv \text{SQL}_{\text{Alg}}$)**.** *Let $[\![\_]\!]_{db}$ be a* well-sorted *database instance and sq be a SQL_{Coq} query, aq a SQL_{Alg} query then:*

$$\forall \mathscr{E}, sq, \mathbb{W}^{\text{q}}(sq) \implies [\![\mathbb{T}^{\text{q}}(sq)]\!]^{\text{Q}}_{\mathscr{E}} = [\![sq]\!]^{\text{q}}_{\mathscr{E}}$$
$$\forall \mathscr{E}, aq,\ [\![\mathbb{T}^{\text{Q}}(aq)]\!]^{\text{q}}_{\mathscr{E}} = [\![aq]\!]^{\text{Q}}_{\mathscr{E}}$$

The proof proceeds by (mutual) structural induction over queries and formulae. Actually the proof is made by induction over the sizes of queries and formulae. It consists of 500 lines of Coq code and heavily relies on a tactic which allows to automate the proofs that size for sub-objects is decreasing. For the correctness of $\mathbb{T}^{\text{q}}(\_)$, the well-formedness hypothesis of the theorem essentially ensures that Cartesian product and natural join coincide. What was interesting is that the well-formedness hypothesis was mandatory and this sheds light on the fact that, indeed, SQL `from` behaves as a cross product. For both translations, well-sortedness ensures that reasoning over tuples' labels in the evaluation of a query can be made globally, by "statically" computing the labels over a query.

5 Conclusions

Seeking a formal semantics for SQL has been a longstanding quest for the database community. In this article, we presented a formal, Coq *mechanised*, *executable* semantics for a large realistic fragment of SQL.

In an early version of the development, we defined a pure set-theoretic semantics and only addressed the SQL's fragment with no duplicates. Then we addressed the bag aspects of SQL and were pleasantly surprised to discover that adding them was not so problematic. Therefore, the widespread belief that *the* problem for SQL is to assign it a bag semantics is not as crucial as it seemed to be. Also, grasping `NULL`'s semantics is often considered one of the most difficult aspects to address, this is due to the fact that SQL does not treat them uniformly according to the context. We handled `NULL`'s thanks to a 3-valued logic. What was *really challenging* was to accurately and faithfully handle *correlated* sub-queries. Particularly tricky was to grasp SQL's management of expressions and environments in the presence of such queries. The ISO/IEC document was of little help along this path. On the contrary, Coq was an enlightening, very demanding master of invaluable help in defining semantically relevant

[10] For algebraic formulae, the domain parameter `dom` is actually algebraic queries.

$$
\begin{array}{rcl}
\mathrm{Q} & ::= & \textbf{Q\_empty\_tuple} \\
& | & \mathit{table} \\
& | & \mathrm{Q}\ (\textbf{union} \mid \textbf{intersect} \mid \textbf{except})\ \mathrm{Q} \\
& | & \mathrm{Q} \bowtie \mathrm{Q} \\
& | & \pi_{(\overline{e^a\ \textbf{as}\ \text{attribute}})}(\mathrm{Q}) \\
& | & \sigma_{\text{formula}}(\mathrm{Q}) \\
& | & \gamma_{(\overline{e^a\ \textbf{as}\ \text{attribute}}, \overline{e^f}, \text{formula})}(\mathrm{Q})
\end{array}
$$

```
Inductive query : Type :=
  | Q_Empty_Tuple : query
  | Q_Table : relname → query
  | Q_Set : set_op → query → query → query
  | Q_NaturalJoin : query → query → query
  | Q_Pi : list select → query → query
  | Q_Sigma : sql_formula query → query → query
  | Q_Gamma : list select → list funterm → sql_formula query →
              query → query.
```

Figure 9. $\mathrm{SQL}_{\mathrm{Alg}}$ Syntax

```
Definition Q_Delta q :=
let s := sort q in
Q_Gamma
  (map (fun a ⇒ Select_As (A_Expr (F_Dot a)) a)
       (Fset.elements _ s))
  (map (fun a ⇒ F_Dot a) (Fset.elements _ s))
  (Sql_True _) q.

Definition Q_SemiJoin f q1 q2 :=
let sort_q1 :=
   map (fun a ⇒ Select_As (A_Expr (F_Dot a)) a)
       (Fset.elements _ (sort q1)) in
Q_NaturalJoin q1
     (Q_Delta
         (Q_Pi sort_q1
            (Q_Sigma f (Q_NaturalJoin q1 q2)))).

Definition Q_AntiJoin f q1 q2 :=
  Q_Set Diff q1 (Q_SemiJoin f q1 q2).
```

```
Hypothesis null : value.

Definition Q_LeftOuterJoin f q1 q2 :=
let s := Fset.diff _ (sort q2) (sort q1) in
let s' :=
    map (fun a  ⇒ Select_As (A_Expr (F_Constant null)) a)
        (Fset.elements _ s) in
 Q_Set Union
       (Q_Sigma f (Q_NaturalJoin q1 q2))
       (Q_NaturalJoin (Q_AntiJoin f q1 q2)
                      (Q_Pi s' Q_Empty_Tuple)).

Definition Q_OuterJoin f q1 q2 :=
let s := Fset.diff _ (sort q1) (sort q2) in
let s' :=
    map (fun a ⇒ Select_As (A_Expr (F_Constant null)) a)
        (Fset.elements _ s) in
 Q_Set Union
       (Q_LeftOuterJoin f q1 q2)
       (Q_NaturalJoin (Q_AntiJoin f q2 q1)
                      (Q_Pi s' Q_Empty_Tuple)).
```

Figure 10. $\mathrm{SQL}_{\mathrm{Alg}}$ Derived Operators

$$
\begin{array}{l}
[\![\textbf{Q\_empty\_tuple}]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|()|\!\} \\
[\![tbl]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![tbl]\!]_{db} \qquad \text{if } tbl \text{ is a table} \\
[\![q_1\ \textbf{union}\ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \\
[\![q_1\ \textbf{intersect}\ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \\
[\![q_1\ \textbf{except}\ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \\
[\![q_1 \bowtie q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = \\
\left\{\!\!\left| \left(\overline{a_i = c_i}, \overline{b_j = d_j}\right) \right| \begin{array}{l} (\overline{a_i = c_i}) \in [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\overline{b_j = d_j}) \in [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\forall\ i, j,\ a_i = b_j \implies c_i = d_j) \end{array} \right\}\!\!\right\}
\end{array}
$$

$$
\begin{array}{l}
[\![\pi_{(\overline{e_i\ \textbf{as}\ a_i})}(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|(\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(t),[],[t])::\mathscr{E}}}) \mid t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}|\!\} \\
[\![\sigma_f(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \mid [\![f]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top|\!\} \\
[\![\gamma_{(\overline{e_j\ \textbf{as}\ a_j}, \overline{e_i}, f)}(q)]\!]^{\mathrm{q}}_{\mathscr{E}} = \\
\qquad \left\{\!\!\left| \overline{\left(a_j = [\![e_j]\!]^{\mathrm{a}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}}\right)} \right| T \in \mathbb{F}_3 \right\}\!\!\right\} \\
\qquad \text{and } \mathbb{F}_2 \text{ is a partition of } [\![q]\!]^{\mathrm{Q}}_{\mathscr{E}} \text{ according to } \overline{e_i} \\
\qquad \text{and } \mathbb{F}_3 = \left\{\!\!\left| T \in \mathbb{F}_2 \right| [\![f]\!]^{\mathrm{b}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}} = \top \right\}\!\!\right\}
\end{array}
$$

Figure 11. $\mathrm{SQL}_{\mathrm{Alg}}$ Semantics

queries. Such a set of queries, augmented with others not listed in this article, could serve as a benchmark for testing SQL's other implementations.

Thanks to our formal semantics we have been able to relate $\mathrm{SQL}_{\mathrm{Coq}}$ and $\mathrm{SQL}_{\mathrm{Alg}}$ establishing, the first, to our best knowledge, equivalence result for that SQL fragment. Moreover, by doing so, we recover the well-known algebraic equivalences presented in textbooks upon which are based most of optimisations used in practice. Such equivalences were proven, using Coq, in [3]. 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, the fact of adding more features along the time in the standardisation process, seriously, and sadly, departed it from its original elegant foundations. By formally relating SQL and an extended relational algebra, we, humbly, also wanted to pay tribute to the pionneers that designed the foundational aspects of RDBMS's.

Our long term goal is to provide a Coq verified compiler for SQL. The work presented in this article allows to obtain a certified semantic analyser that we plan to extend to features like `order by`. In [4] we provided a certification of the physical layer of a SQL engine where mainstream physical operators such as `sequential scans`, `nested loop joins`,

$$\mathbb{T}^{\mathrm{q}}(tbl) = tbl$$
$$\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ union } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ union } \mathbb{T}^{\mathrm{q}}(q_2)$$
$$\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ intersect } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ intersect } \mathbb{T}^{\mathrm{q}}(q_2)$$
$$\mathbb{T}^{\mathrm{q}}(q_1 \textbf{ except } q2) = \mathbb{T}^{\mathrm{q}}(q_1) \textbf{ except } \mathbb{T}^{\mathrm{q}}(q_2)$$
$$\mathbb{T}^{\mathrm{q}}(\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w) = \pi_{\overline{(e_i \textbf{ as } a_i)}}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)}))$$
$$\mathbb{T}^{\mathrm{q}}(\textbf{select } \overline{e_i \textbf{ as } a_i} \textbf{ from } \overline{f_i} \textbf{ where } w \textbf{ group by } G \textbf{ having } h) = \gamma_{\overline{(e_i \textbf{ as } a_i}, G, \mathbb{T}^{\mathrm{f}}(h))}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)}))$$
$$\mathbb{T}^{\mathrm{from}}(\overline{q(a_i \textbf{ as } b_i)}) = \pi_{\overline{(a_i \textbf{ as } b_i)}}(\mathbb{T}^{\mathrm{q}}(q))$$

$$\mathbb{T}^{\mathrm{Q}}(\textbf{Q\_empty\_tuple}) = \textbf{select } [] \textbf{ from } [\textbf{default\_table}(\overline{a \textbf{ as } a})]$$
$$\mathbb{T}^{\mathrm{Q}}(tbl) = tbl$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ union } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ union } \mathbb{T}^{\mathrm{Q}}(q_2)$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ intersect } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ intersect } \mathbb{T}^{\mathrm{Q}}(q_2)$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \textbf{ except } q2) = \mathbb{T}^{\mathrm{Q}}(q_1) \textbf{ except } \mathbb{T}^{\mathrm{Q}}(q_2)$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \bowtie q_2) = \textbf{select } (\overline{a'_1 \textbf{ as } a_1}_{a_1 \in \ell(q_1)}, \overline{a'_2 \textbf{ as } a_2}_{a_2 \in \ell(q_2) \setminus \ell(q_1)}) \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q_1)(\overline{a_1 \textbf{ as } a'_1}); \mathbb{T}^{\mathrm{Q}}(q_2)(\overline{a_2 \textbf{ as } a'_2})] \textbf{ where } \overline{(a'_1 = a'_2)}_{a_1 \in \ell(q_1), a_2 \in \ell(q_2), a_1 = a_2}$$
where $\overline{a'_1}$ and $\overline{a'_2}$ are fresh names
$$\mathbb{T}^{\mathrm{Q}}(\pi_{\overline{(e \textbf{ as } a)}}(q)) = \textbf{select } (\overline{e \textbf{ as } a}) \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})]$$
$$\mathbb{T}^{\mathrm{Q}}(\sigma_f(q)) = \textbf{select } * \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})] \textbf{where } \mathbb{T}^{\mathrm{F}}(f)$$
$$\mathbb{T}^{\mathrm{Q}}(\gamma_{\overline{(e \textbf{ as } a}, G, f)}(q)) = \textbf{select } (\overline{e \textbf{ as } a}) \textbf{ from } [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \textbf{ as } a})] \textbf{ group by } G \textbf{ having } \mathbb{T}^{\mathrm{F}}(f)$$

Figure 12. Translations between $\mathrm{SQL}_{\mathrm{Coq}}$ and $\mathrm{SQL}_{\mathrm{Alg}}$.

`index joins` or `bitmap index joins` are formally specified and implemented. What remains to be done is to address the logical optimisation part of the compiler.

References

[1] T. Arvin. 2017. *Comparison of different SQL's implementations.* http://troels.arvin.dk/db/rdbms

[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. 2017. Handling Environments in a Nested Relational Algebra with Combinators and an Implementation in a Verified Query Compiler. In *SIGMOD Conference, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu (Eds.). ACM, 1555–1569.

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

[4] V. Benzaken, É. Contejean, C. Keller, and E. Martins. 2018. A Coq formalisation of SQL's execution engines. In *International Conference on Interactive Theorem Proving (ITP 2018)*. Oxford, United Kingdom.

[5] S. Ceri and G. Gotlob. 1985. Translating SQL into Relational Algebra: Optimisation, Semantics, and Equivalence of SQL Queries. *IEEE Trans., on Software Engineering* SE-11 (April 1985), 324–345.

[6] S. Chu, K. Weitz, A. Cheung, and D. Suciu. 2017. HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics. In *PLDI 2017*. ACM, New York, NY, USA, 510–524.

[7] S. Cluet and G. Moerkotte. 1993. Nested Queries in Object Bases. In *Database Programming Languages (DBPL-4), Manhattan, New York City, USA, 30 August - 1 September 1993*. 226–242.

[8] H. Garcia-Molina, J D. Ullman, and J. Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[9] C. Gonzalia. 2003. Towards a Formalisation of Relational Database Theory in Constructive Type Theory. In *RelMiCS (LNCS)*, R. Berghammer, B. Möller, and G. Struth (Eds.), Vol. 3051. Springer, 137–148.

[10] C. Gonzalia. 2006. *Relations in Dependent Type Theory.* Ph.D. Dissertation. Chalmers Göteborg University.

[11] T J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. 31–40.

[12] P. Guagliardo and L. Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *PVLDB* 11, 1 (2017), 27–39.

[13] R. Harper. 2016. *Practical Foundations for Programming Languages.* Cambridge University Press.

[14] ISO/IEC. 2006. Information technology - Database Languages - SQL - Part 2: Foundation (SQL/Foundation). Final Commitee Draft.

[15] X. Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446.

[16] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. 2010. Toward a Verified Relational Database Management System. In *ACM Int. Conf. POPL*.

[17] M. Negri, G. Pelagatti, and L. Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* 16, 3 (1991), 513–534.

[18] B. Pierce et al. 2018. *Software Foundations - Programming Languages Foundations*. Vol. 2.

[19] The Agda Development Team. 2010. *The Agda Proof Assistant Reference Manual.* http://wiki.portal.chalmers.se/agda/pmwiki.php

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

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