Vers un traitement certifié des données : SQL à l'épreuve de COQ

Toward a Coq verified SQL's compiler

Véronique Benzaken, Évelyne Contejean, Chantal Keller

Vals Research group - LRI - CNRS - Université Paris Saclay

JIRC, February 6, 2020

Datacert : Projet ANR 15-CE39-0009

Motivations

Data are pervasive and valuable ...

Stored into relational database systems ... most widespread ones

Mature implementations

Oracle, DB_2 IBM, SQLServer, Postgresql, MySql, SQLite ...

SQL *the* standard programming language for such systems

... Little attention to guarantee such systems are reliable and safe.

provide a Coq verified SQL's compiler

SQL's compilation

Syntactic analysis — SQL $\rightarrow$ AST

Semantic analysis — AST $\rightarrow$ AST_{sem}

Textbooks	$AST_{sem} \equiv$	relational algebra expression
Real life		depends on DB vendors

Optimisation / Query planning — $AST_{sem} \rightarrow AST_{sem} \rightarrow AST_{phys}$

Logical $AST_{sem} \rightarrow AST_{sem}$
rewritings / algebraic equivalences

Physical $AST_{sem} \rightarrow AST_{phys}$
auxiliary data structures (B trees, Hash tables etc)
physical algebra – different implementations of operators
data dependent statistics

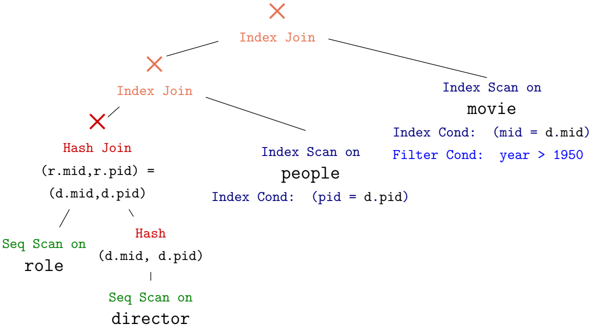
SQL's compilation: example

```
select lastname from people p, director d, role r, movie m
where  d.mid = r.mid and d.pid = r.pid and p.pid = d.pid and
       m.mid = d.mid and m.year > 1950;
```

$$\pi_{\texttt{lastname}}(\sigma_{\texttt{year}>1950}(\texttt{people} \bowtie \texttt{director} \bowtie \texttt{role} \bowtie \texttt{movie}))$$

SQL's compilation: example

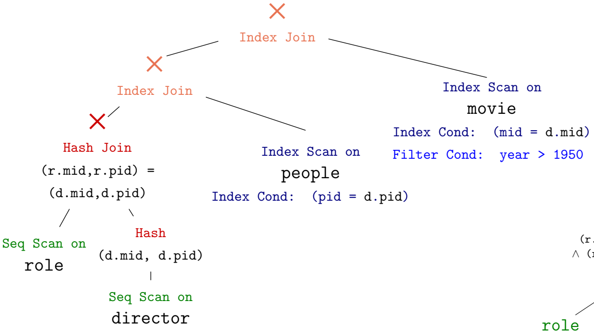
```
explain(
select lastname from people p, director d, role r, movie m
where  d.mid = r.mid and d.pid = r.pid and p.pid = d.pid and
       m.mid = d.mid and m.year > 1950;)
```

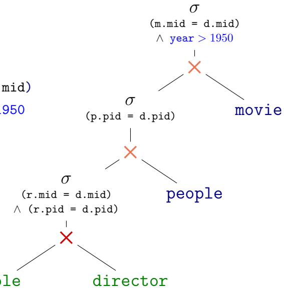


$$\pi_{\texttt{lastname}}(\sigma_{\texttt{year}>1950}(\texttt{people} \bowtie \texttt{director} \bowtie \texttt{role} \bowtie \texttt{movie}))$$

SQL's compilation: example

```
explain(
select lastname from people p, director d, role r, movie m
where  d.mid = r.mid and d.pid = r.pid and p.pid = d.pid and
       m.mid = d.mid and m.year > 1950;
```





$$\pi_{\texttt{lastname}}(\sigma_{\texttt{year}>1950}(\texttt{people} \bowtie \texttt{director} \bowtie \texttt{role} \bowtie \texttt{movie}))$$

General goal

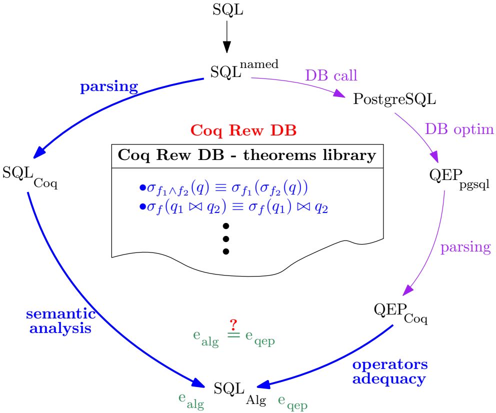
For any SQL query

guarantee with the strongest (a.k.a Coq) possible ensurance

that

system produced evaluation strategy preserves query's semantics.

Precisely



? { normalize w.r.t. Coq Rew DB
then check equivalence

validity :
check e_{alg} e_{qep} = true

$\Rightarrow \forall$ db. eval db e_{alg} = eval db e_{qep}

Roadmap

1. Equip SQL with a formal, executable semantics
2. Rigorously relate it with relational algebra

recover well-known algebraic rewritings

30 years research efforts

[Ceri&al 85, Negri&al 91, Malecha&al 10]

[Guagliardo&al 17, Auerbach&al 17, Chu&al 17]

by-product Extracted certified semantic analyser

3. Coq specification of data-centric operators
4. Physical and Relational algebras as specification's refinements
5. A Coq proven logical rewritings / algebraic equivalences library

Basics

Relational model

model information through relations

r(a, b) schema

r relation's name, a, b attributes, {a, b} sort

denote finite sets of tuples

position vs name

$r = \{(1, 2); (3, 2); (1, 1)\}$

$r = \{t1; t2; t3\}$

$$t1(a) = 1, \quad t1(b) = 2$$
$$t2(a) = 3, \quad t2(b) = 2$$
$$t3(a) = 1, \quad t3(b) = 1$$

extract information through query langages

relational algebra: λ-calculus for databases

Relational model

model information through relations

r(a, b) schema

r relation's name, a, b attributes, {a, b} sort

denote finite sets of tuples

position vs name

$r = \{(1, 2); (3, 2); (1, 1)\}$

$r = \{t1; t2; t3\}$

$t1.a = 1, \quad t1.b = 2$
$t2.a = 3, \quad t2.b = 2$
$t3.a = 1, \quad t3.b = 1$

extract information through query langages

relational algebra: λ-calculus for databases

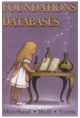
Relational algebra - syntax

position (SPC)

$$q \;:=\; r \mid \sigma_f(q) \mid \pi_W(q) \mid q \times q$$
$$\mid\; q \cup q \mid q \cap q \mid q \setminus q$$

named (SPJR)

$$q \;:=\; r \mid \sigma_f(q) \mid \pi_W(q) \mid \rho_g(q) \mid q \bowtie q$$
$$\mid\; q \cup q \mid q \cap q \mid q \setminus q$$



denotable attributes

⇒ named perspective

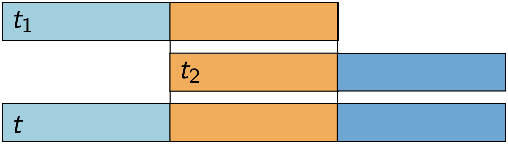
Named relational algebra - semantics

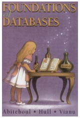
$$[\![\sigma_f(q)]\!] = \{t \in [\![q]\!] \mid [\![f]\!](t)\}$$

$$[\![\pi_W(q)]\!] = \{t|_W \mid t \in [\![q]\!]\}$$

$$[\![\rho_g(q)]\!] = \{t' \mid \exists t \in [\![q]\!], \forall a \in sort(q), t'.g(a) = t.a\}$$

$$[\![q_1 \bowtie q_2]\!] = \{t \mid \exists t_1 \in [\![q_1]\!], \exists t_2 \in [\![q_2]\!], t|_{sort(q_1)} = t_1 \wedge t|_{sort(q_2)} = t_2\}$$





$$sort(q_1) \cap sort(q_2) = \emptyset \qquad \bowtie \;\equiv\; \times$$

SQL: a simple declarative language

SQL "inter-galactic" dialect for manipulating (relational) data

Declarative DSL describe what opposed as how

```
select expression
from query
where condition
group by expression
having condition
```

With attribute's names as first-class citizens

⇒ name-based perspective

SQL informal semantics

```
select expression
from query
where condition
group by expression
having condition
```

Evaluates the `from` ($\bowtie$)

Filters with `where` (σ)

Builds groups with `group by expression` (?)

Discards groups not satisfying `having` (?)

Evaluates `select` expressions ($\pi + \rho$)

SQL : simple and declarative

$$r_1 = \left\{ \begin{array}{c} (a=1,b=4,c=1);(a=1,b=5,c=1); \\ (a=7,b=4,c=3);(a=7,b=1,c=1) \end{array} \right\}$$

$$r_2 = \{(c=1,d=4);(c=7,d=4)\}$$

```
select * from r1, r2 where b>d ;
```

position

$\sigma_{\mathtt{b>d}}(\mathtt{r_1} \times \mathtt{r_2})$

$\neq \sigma_{\mathtt{b>d}}(\mathtt{r_1} \bowtie \mathtt{r_2})$

$$\{\!|(a=1,b=5,c=1,c=1,d=4);(a=1,b=5,c=1,c=7,d=4)|\!\}$$

```
select a from r1 where exists (select d from r2 where a < d);
```

exists: test for non emptyness ($\emptyset$)

nested, correlated query

$\{\!|(a=1);(a=1)|\!\}$

bag

$\pi_{\{\mathtt{a}\}}(\sigma_{\mathtt{a<d}}(\mathtt{r_1} \bowtie \mathtt{r_2}))$

SQL : simple and declarative

$$r_1 = \left\{ \begin{array}{c} (a = 1, b = 4, c = 1); (a = 1, b = 5, c = 1); \\ (a = 7, b = 4, c = 3); (a = 7, b = 1, c = 1) \end{array} \right\}$$

$$r_2 = \{(c = 1, d = 4); (c = 7, d = 4)\}$$

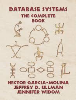
```
select b, sum(a) from r1 where b ≠ 5 group by b;
```

$$\{\!|(a = 1, b = 4, c = 1); (a = 7, b = 4, c = 3); (a = 7, b = 1, c = 1)|\!\}$$

a	b	c
1 7	4	1 3
7	1	1

$\rightsquigarrow$

b	sum(a)
4	8
1	7

In 

$\gamma_{\mathtt{b},sum(\mathtt{a})}(\sigma_{\mathtt{b}\neq 5}(\mathtt{r}_1))$

SQL : simple and declarative

```
select b, 2*(a+c), sum(a) from r1 where a+b = 7
        group by b, a+c having avg(b+c) > 6;
```

No algebraic equivalent in textbooks

SQL: recap

based on relational algebra for `select-from-where`

mixes both perspectives: named and position

enjoys bag semantics

not set semantics

More importantly

allows for definition of complex expressions and aggregates

in the presence of `NULL` values
representing incomplete information
handled thanks to a 3-valued logic `unknown`

with nested and correlated queries
$\Longrightarrow$ surprising behaviours due to SQL's environment management

Mechanised Semantics: Roadmap

1. `NULL`'s
2. Understand SQL's environment management and evaluation
3. Define SQL's Coq mechanised semantics: SQL_{Coq}
4. Define a Coq mechanised bag algebra: SQL_{Alg}
5. Prove the equivalence $SQL_{Coq} \equiv SQL_{Alg}$

Mechanised semantics

SQL: NULL's

NULL absorbing element in expressions $\texttt{NULL} + 2 \rightsquigarrow \texttt{NULL}$

NULL uncomparable with anything else

$$(\texttt{NULL} = \texttt{NULL}) \rightsquigarrow \bot$$
$$(\texttt{NULL} \neq \texttt{NULL}) \rightsquigarrow \bot$$
$$(\texttt{NULL} \neq 1) \rightsquigarrow \bot$$

NULL equals NULL to form groups (`group by`)

SQL: aggregates, nesting, correlation

$$r_1 = \{\!|(a_1 = 1, b_1 = i) \mid 1 \leq i \leq 10|\!\} \cup \{\!|(a_1 = 2, b_1 = i) \mid 1 \leq i \leq 10|\!\} \cup \{\!|(a_1 = 3, b_1 = i) \mid 1 \leq i \leq 5|\!\} \cup \{\!|(a_1 = 4, b_1 = i) \mid 6 \leq i \leq 10|\!\}$$

$$r_2 = \{\!|(a_2 = 7, b_2 = 7), (a_2 = 7, b_2 = 7)|\!\}$$

$Q(k)$:
```
select a1 from r1 group by a1 having
    exists (select a2 from r2 group by a2
            having expression with aggregates = k);
```

How to evaluate `expression`? On which groups?

4 a_1-groups for r_1

$$\begin{array}{lcll} \mathcal{G}_1 & = & \{\!|(a_1 = 1, b_1 = i) \mid 1 \leq i \leq 10|\!\} & \text{cardinality} = 10 \\ \mathcal{G}_2 & = & \{\!|(a_1 = 2, b_1 = i) \mid 1 \leq i \leq 10|\!\} & \text{cardinality} = 10 \\ \mathcal{G}_3 & = & \{\!|(a_1 = 3, b_1 = i) \mid 1 \leq i \leq 5|\!\} & \text{cardinality} = 5 \\ \mathcal{G}_4 & = & \{\!|(a_1 = 4, b_1 = i) \mid 6 \leq i \leq 10|\!\} & \text{cardinality} = 5 \end{array}$$

a single a_2-group for r_2

$$\mathcal{G}' = \{\!|(a_2 = 7, b_2 = 7), (a_2 = 7, b_2 = 7)|\!\} \quad \text{cardinality} = 2$$

SQL: aggregates, nesting, correlation

$Q_1(k)$: `expression` = `sum(1+0*b2)`, computes group's cardinality

$$k = 2 \leadsto \{\!|(a_1 = 1); (a_1 = 2); (a_1 = 3); (a_1 = 4)|\!\}$$

$$k \neq 2 \leadsto \{\!|\ |\!\}$$

groups with cardinality 2 $\leadsto$ $\mathcal{G}'$, a_2-group of r_2

SQL: aggregates, nesting, correlation

$Q_2(k)$: `expression` = `sum(1)`, again, computes group's cardinality

$$k = 2 \rightsquigarrow \{\!|(a_1 = 1); (a_1 = 2); (a_1 = 3); (a_1 = 4)|\!\}$$

$$k \neq 2 \rightsquigarrow \{\!|\ |\!\}$$

groups with cardinality 2 $\rightsquigarrow$ $\mathcal{G}'$, a_2-group of r_2

SQL: aggregates, nesting, correlation

$Q_2(k)$: `expression` = `sum(1)`, again, computes group's cardinality

$$k = 2 \leadsto \{\!|(a_1 = 1); (a_1 = 2); (a_1 = 3); (a_1 = 4)|\!\}$$
$$k \neq 2 \leadsto \{\!|\ |\!\}$$

groups with cardinality 2 $\leadsto$ $\mathcal{G}'$, a_2-group of r_2

Tentative conclusion: `1+0*b2 = 1`

SQL: aggregates, nesting, correlation

$Q_3(k)$: `expression` $=$ `sum(1+0*b1)`

$$k = 5 \rightsquigarrow \{\!|(a_1 = 3); (a_1 = 4)|\!\} \quad k = 10 \rightsquigarrow \{\!|(a_1 = 1); (a_1 = 2)|\!\}$$

$$k \neq 5 \wedge k \neq 10 \rightsquigarrow \{\!|\ |\!\}$$

groups with cardinality 5 and 10 $\rightsquigarrow$ $\mathcal{G}_i$, a_1-group of r_1

SQL: aggregates, nesting, correlation

$Q_3(k)$: `expression` = `sum(1+0*b1)`

$$k = 5 \rightsquigarrow \{\!|(a_1 = 3); (a_1 = 4)|\!\} \quad k = 10 \rightsquigarrow \{\!|(a_1 = 1); (a_1 = 2)|\!\}$$

$$k \neq 5 \wedge k \neq 10 \rightsquigarrow \{\!|\ |\!\}$$

groups with cardinality 5 and 10 $\rightsquigarrow$ $\mathcal{G}_i$, a_1-group of r_1

Tentative conclusion: `1+0*b2 = 1 <> 1+0*b1`

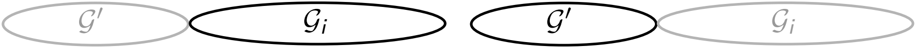
SQL: aggregates, nesting, correlation

$Q_4(k)$: `expression` = `sum(1+0*b1) + sum(1+0*b2)`

$$k = 7 \leadsto \{\!|(a_1 = 3); (a_1 = 4)|\!\} \quad k = 12 \leadsto \{\!|(a_1 = 1); (a_1 = 2)|\!\}$$

$$k \neq 7, k \neq 12 \leadsto \{\!|\ |\!\}$$

$7 = 5 + 2$ and $12 = 10 + 2$



Different sub-expressions of the same expression evaluated in different environments !!!

SQL: aggregates, nesting, correlation

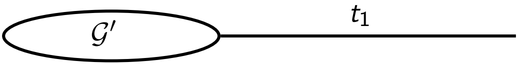
$Q_5(k)$: `expression = sum(1+1*a1+0*b2)`

$$k = 4 \rightsquigarrow \{\!|(a_1 = 1)|\!\} \quad k = 6 \rightsquigarrow \{\!|(a_1 = 2)|\!\}$$

$$k = 8 \rightsquigarrow \{\!|(a_1 = 3)|\!\} \quad k = 10 \rightsquigarrow \{\!|(a_1 = 4)|\!\}$$

$$k \neq 4 \wedge k \neq 6 \wedge k \neq 8 \wedge k \neq 10 \rightsquigarrow \{\!|\ |\!\}$$

`sum(1+1*a1+0*b2)` evaluated for each `a2`-group
combined with projection of `a1`-group



t_1 = projection of $\mathcal{G}_i$ on $a_1 \rightsquigarrow t_1 = (a_1 = i)$
`1+1*a1+0*b2` is evaluated over tuples $t' \bowtie t_1$, where $t' \in \mathcal{G}'$ $\rightsquigarrow$ `1+i`
$\mathcal{G}'$ has cardinality 2, $\rightsquigarrow$ `sum(1+1*a1+0*b2) = 2*(1+i)`

SQL: aggregates, nesting, correlation

$Q_6(k)$: `sum(1+0*b1+0*b2)`

```
ERROR: subquery uses ungrouped column "r1.b1" from outer query
LINE 1: ...sts (select a2 from r2 group by a2 having sum(1+0*b1+0*b2) =
```

$Q_7(k)$: `sum(1+0*b1+0*a2)`

```
ERROR: subquery uses ungrouped column "r1.b1" from outer query
LINE 1: ...sts (select a2 from r2 group by a2 having sum(1+0*b1+0*a2) =
```

Environments: recap

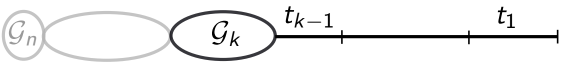
A stack of slices, levels of nesting, innermost on top

$$[S_n; S_{n-1}; \ldots; S_i,; \ldots; S_1]$$

$S = (A, g, \mathcal{G})$ = (attributes, grouping expressions, groups of tuples)

Evaluation: recap

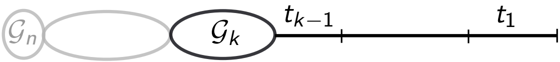
- simple expression $\rightsquigarrow$ (unique) binding for each attribute
- `function`($\overline{e}$) $\rightsquigarrow$ evaluate independently each e_i of ($\overline{e}$)
- `aggregate(cst)` $\rightsquigarrow$ use innermost slice (cardinality)
- `aggregate`(e) in $[S_n; \ldots; S_1]$
  $\rightsquigarrow$ find the smallest suitable suffix $[S_k; S_{k-1}, ; \ldots; S_1]$
  s.t. e is built upon $A(S_k) \cup g(S_{k-1}) \cup \ldots \cup g(S_1)$
  split tuples of kth slice



Evaluation: recap

SQHeLL

- simple expression ⇝ (unique) binding for each attribute
- function($\overline{e}$) ⇝ evaluate independently each e_i of ($\overline{e}$)
- aggregate(cst) ⇝ use innermost slice (cardinality)
- aggregate(e) in $[S_n; \ldots; S_1]$
  ⇝ find the smallest suitable suffix $[S_k; S_{k-1}, ; \ldots ; S_1]$
  s.t. e is built upon $A(S_k) \cup g(S_{k-1}) \cup \ldots \cup g(S_1)$
  split tuples of kth slice



SQL_{Coq} Syntax

Queries

```
Inductive set_op := Union | Intersect | Except.
Inductive select := Select_As : aggterm → attribute → select.
Inductive select_item := Select_Star | Select_List : list select → select_item.
Inductive group_by := Finest_P | Group_By : list funterm → group_by.

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

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

with from_item := From_Item : sql_query → att_renaming_item → sql_from_item.
```

where, group by and having mandatory in SQL_{Coq}

no where in SQL ⇝ TTrue

no group by but having in SQL ⇝ Group_By nil

no group by nor having in SQL ⇝ Finest_P+ TTrue

SQL$_{Coq}$ Syntax

Formulas

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

Inductive formula (dom : Type) :=
  | Conj : conjunct → formula dom → formula dom → formula dom
  | Not : formula dom → formula dom
  | TTrue : 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.
```

FO + `in` + `exists`

$\forall \rightsquigarrow$ `all` ; $\exists \rightsquigarrow$ `any`

`in` (membership) $\rightsquigarrow$ $\_ \in \_$ (not a usual predicate over values)

`exists` $\rightsquigarrow$ non emptyness

parameterised by `dom`

finite domain of interpretation

Mechanised semantics

Formulas

```
Hypothesis B : Bool.Rcd. (* parametric Booleans *)
Hypothesis I : env_type → dom → bagT (* bags of tuples *).
Fixpoint eval_formula env f : Bool.b B := match f with
  | Conj a f1 f2 ⇒ (interp_conj B a) (eval_formula env f1) (eval_formula env f2)
  | Not f ⇒ Bool.negb B (eval_formula env f)
  | TTrue ⇒ Bool.true B
  | Pred p l ⇒ interp_predicate p (map (interp_aggterm env) l)
  | Quant l p qtf sq ⇒ let lt := map (interp_aggterm env) l in
    interp_quant B qtf (fun x ⇒ let la := Fset.elements _ (labels T x) in
                                      interp_predicate p (lt ++ map (dot T x) la))
                 (Febag.elements _ (I env sq))
  | In s sq ⇒ let p := (projection env (Select_List s)) in
    interp_quant B Any
       (fun x ⇒ match Oeset.compare (OTuple T) 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))
  | Exists sq ⇒ if Febag.is_empty _ (I env sq) then Bool.false B else Bool.true B
  end.
```

(`In s sq`) translates into $\exists x, x \in sq \wedge x = \mathit{projection\ env\ s}$

In presence of `NULL` in x or in *projection env s* $\rightsquigarrow$ `unknown`

Mechanised semantics

Queries

```
Fixpoint eval_sql_query env (sq : sql_query) {struct sq} :=
match sq with
 | Sql_Table tbl ⇒ instance tbl
 | Sql_Set o sq1 sq2 ⇒ [...]
 | 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 the formula f1, with old names *)
     Febag.filter _
         (fun t ⇒ Bool.is_true B (* casting parametric Booleans to Bool2 *)
                     (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 B (* casting parametric Booleans to Bool2 *)
                    (eval_sql_formula eval_sql_query (env_g env gby g) f2))
              lg1 in
  (** applying the 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
```

evaluate `from`, then filter wrt (casted) `where`, then build groups,
then filter wrt (casted) `having`, then project wrt `select`

Algebra

Relating SQL_{Coq} with relational algebra

Define SQL_{Alg} an extended relational algebra

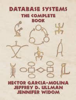
Enjoying a bag semantics and

Natively accounting for `group by having`

SQL_{Alg} syntax

```
Inductive alg_query : Type :=
  | Q_Empty_Tuple : alg_query
  | Q_Table : relname → alg_query
  | Q_Set : set_op → alg_query → alg_query → alg_query
  | Q_Join : alg_query → alg_query → alg_query
  | Q_Pi : list select → alg_query → alg_query
  | Q_Sigma : (formula alg_query) → alg_query → alg_query
  (* extending the usual γ textbook operator *)
  | Q_Gamma :
      (* aggregated (output) expressions *) list select →
      (* grouping expressions *) list funterm →
      (* handling having condition *) (formula alg_query) →
      (* query *) alg_query → alg_query.
```

traditional algebra + γ operator: `Q_Gamma`

extending the one in 

to account for `having`

`formulas` shared with SQL_{Coq}

SQL_{Alg} semantics

```
Fixpoint eval_alg_query env q {struct q} : bagT :=
  match q with
  | Q_Empty_Tuple ⇒ Febag.singleton _ (empty_tuple T)
  | Q_Table r ⇒ instance r
  | Q_Set o q1 q2 ⇒ [...]
  | Q_Join q1 q2 ⇒ natural_join (eval_alg_query env q1) (eval_alg_query env q2)
  | Q_Pi s q ⇒
    Febag.map _ _
       (fun t ⇒ projection (env_t env t) (Select_List s)) (eval_alg_query env q)
  | Q_Sigma f q ⇒
    Febag.filter _
       (fun t ⇒ Bool.is_true B (eval_formula _ eval_alg_query (env_t env t) f))
       (eval_alg_query env q)
  | Q_Gamma s g f q ⇒
    Febag.mk_bag _
       (map (fun l ⇒ projection (env_g env (Group_By g) l) (Select_List s))
       (filter (fun l ⇒ Bool.is_true B
                    (eval_formula _ eval_alg_query (env_g env (Group_By g) l) f))
               (make_groups env (eval_alg_query env q) (Group_By g))))
  end.
```

formulas and environments shared with SQL_{Coq}

$\gamma_{s,g,f}(q)$: evaluate query q, then build groups wrt g, then filter wrt (casted) condition f, then project wrt s

$SQL_{Coq} \equiv SQL_{Alg}$

A database instance $[\![\_]\!]_{db}$ is well-sorted when all tuples in the same relation have the same attributes as the relation's sort

A SQL_{Coq} query sq is well-formed when all attributes in its `from` clause are pairwise distinct (and recursively)

Theorem

Let $[\![\_]\!]_{db}$ *be well-sorted*
Let sq *be a well-formed* SQL_{Coq} *query, then:*

$$\forall env, \quad [\![\mathbb{T}^{q}(sq)]\!]^{Q}_{env} = [\![sq]\!]^{q}_{env}$$

Let aq *be a* SQL_{Alg} *query then:*

$$\forall env, \quad [\![\mathbb{T}^{Q}(aq)]\!]^{q}_{env} = [\![aq]\!]^{Q}_{env}$$

Physical algebra

High-level specification

Iterator interface operators			
data centric operators	**SQL algebra**	**ϕ algebra**	
		simple	**index based**
map	r, π	`Seq Scan`	`Index scan` `Bitmap index scan`
join	$\times$	`Nested loop` `Block nested loop`	`Hash join` `Index join`
filter	σ	`Filter`	
group	γ	`Group`	
		Intermediate results storage operators	
		`Materialize`	

map	iterate
join	combine
filter	extract according to a condition
group	aggregate results

specify these operations high degree of abstraction

Main idea

High-Level Spec

`Definition is_a_..._op p o :=`

$\forall$ x t, nb_occ t (o p x) $= f_{o,p}$(t, nb_occ t x)

ϕ-algebra

`Lemma` ϕ`_..._op_is_a_..._op :`

$H_\phi \Rightarrow \forall$ x t, nb_occ t (o_ϕ p x) $= f_{o,p}$(t, nb_occ t x)

SQL Algebra

`Lemma` SQL`_..._op_is_a_..._op :`

$H_{SQL} \Rightarrow \forall$ x t, nb_occ t (o_{SQL} p x) $= f_{o,p}$(t, nb_occ t x)

Bridge

`Lemma` ϕ`_..._op_implements_SQL_..._op :`

$H_\phi \wedge H_{SQL} \Rightarrow \forall$ x t, nb_occ t (o_ϕ p x) = nb_occ t (o_{SQL} p x)

Example: filter

```
Section F.

Hypothesis elt : Type.
Hypotheses container container' : Type.
Hypothesis nb_occ : elt → container → nat.
Hypothesis nb_occ' : elt → container' → nat.

Definition is_a_filter_op (f: elt → bool) (fltr: container → container') :=
  ∀s t, nb_occ' t (fltr s) = (nb_occ t s) * (if f t then 1 else 0).

End F.
```

Bridging: two filter operators are interchangeable

```
∀s t, nb_occ' t (fltr1 s)
          = (nb_occ t s) * (if f t then 1 else 0)
          = nb_occ' t (fltr2 s)
```

Similar for other operators

Adequacy

```
Fixpoint eval_query env q {struct q} :=
  match q with
    | Q_Sigma f q ⇒
       filter (fun t ⇒ eval_formula (env_t env t) f) (eval_query env q)
    | ...
  end

with eval_formula env f := [ ... ]

end.
```

Adequacy

```
Lemma Q_Sigma_is_a_filter_op : ∀env f,
 is_a_filter_op [...] (* elt := tuple, container, container' := query *)
    (* nb_occ, nb_occ' *) (fun t q ⇒ nb_occ t (eval_query env q)
                          (fun t q ⇒ nb_occ t (eval_query env q)
    (fun t ⇒ eval_formula (env_t env t) f)
    (fun q ⇒ Q_Sigma f q).

(* ∀q t, nb_occ t (eval_query env (Q_Sigma f q)) =
                (nb_occ t (eval_query env q)) *
                (if eval_formula (env_t env t) f then 1 else 0) *)
```

Bridging lemmas

```
Lemma mk_filter_implements_Q_Sigma :
    [...]
    let F := mk_filter elt (fun t ⇒ eval_formula (env_t env t) f) C in
    eval_query env q = materialize C c →
    eval_query env (Q_Sigma f q) = materialize F c.
```

```
Lemma NL_implements_Q_Join :
  (* Provided that the sorts are disjoined... *)
  ∀C1 C2 env q1 q2, (sort q1 interS sort q2) = emptysetS →
    (∀t, 0 < nb_occ t (eval_query env q1) → support t = sort q1) →
    (∀t, 0 < nb_occ t eval_query env q2 → support t = sort q2) →
    let NL := NestedLoop.build [...] C1 C2 in
    ∀c1 c2, (* ... if the two cursors implement the queries... *)
    (∀t, nb_occ t (eval_query env q1) = List.nb_occ t (materialize C1 c1)) →
    (∀t, nb_occ t (eval_query env q2) = List.nb_occ t (materialize C2 c2)) →
      (* ... then the nested loop implements the join *)
      ∀t, nb_occ t (eval_query env (Q_Join q1 q2)) =
        List.nb_occ t (materialize NL (NestedLoop.mk_cursor C1 C2 nil c1 c2)).
```

Conjunction of hypotheses justify that physical operators implement a cross-product

QEP_{Coq} a domain specific language for QEP's

QEP_Coq: A QEP language

$$
\begin{array}{lll}
\text{cursor} & ::= & \\
& | & \textbf{SeqScan} \;\; \textit{table} \\
& | & \textbf{Projection} \;\; (\overline{e^a \;\textbf{as}\; \text{attribute}}) \;\; \text{cursor} \\
& | & \textbf{Filter} \;\; \text{formula} \;\; \text{cursor} \\
& | & \textbf{NestedLoop} \;\; \text{cursor} \;\; \text{cursor} \\
& | & \textbf{ThetaNestedLoop} \;\; \text{formula} \;\; \text{cursor} \;\; \text{cursor} \\
& | & \textbf{Group} \;\; \overline{e^f} \;\; \text{formula} \;\; (\overline{e^a \;\textbf{as}\; \text{attribute}}) \;\; \text{cursor} \\
& | & \textbf{IndexScan} \;\; \text{index} \;\; \overline{\text{value}} \\
& | & \textbf{IndexJoin} \;\; \overline{e^f} \;\; \text{cursor} \;\; \text{index}
\end{array}
$$

$$
\begin{array}{lll}
\text{index} & ::= & \\
& | & \textbf{FilterIndex} \;\; \textit{table} \;\; \overline{e^f} \;\; \overline{p} \qquad p \in \textit{predicate} \\
& | & \textbf{HashIndex} \;\; \text{cursor} \;\; \overline{e^f}
\end{array}
$$

$QEP_{Coq} \equiv SQL_{Alg}$

Theorem ($QEP_{Coq} \equiv SQL_{Alg}$)

Given a well-sorted *database instance and q a QEP_{Coq}, then:*

$$\mathbb{W}^{QEP}(q) \implies [\![\mathbb{T}^{QEP}(q)]\!]^{Q}_{\emptyset} = [\![q]\!]^{QEP}$$

$\mathbb{W}^{QEP}()$ well formed condition for QEP's

similar to well-formedness for SQL_{Alg} queries

$\mathbb{T}^{QEP}(q)$

translation from QEP's to SQL_{Alg} expressions

Coq $^{rew}_{DB}$: a DB rewritings library

Textbooks' equivalences

$$\sigma_{f_1 \wedge f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) \quad (1)$$

$$\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q)) \quad (2)$$

$$(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3) \quad (3)$$

$$q_1 \bowtie q_2 \equiv q_2 \bowtie q_1 \quad (4)$$

$$\pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) \quad \text{if } W_1 \subseteq W_2 \quad (5)$$

$$\pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q)) \quad \text{if } \mathcal{A}tt(f) \subseteq W \quad (6)$$

$$\sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 \quad \text{if } \mathcal{A}tt(f) \subseteq sort(q_1) \quad (7)$$

$$\sigma_f(q_1 \square q_2) \equiv \sigma_f(q_1) \square \sigma_f(q_2) \quad \text{where } \square \text{ is } \cup \text{ or } \cap \quad (8)$$

Caveat

relational algebra projections

π_W

W, a finite set of attributes

SQL more complex as projections (`select` clause)

involve complex expressions and substitutions

$\pi_{\overline{e \ as \ a}}$.

Need to embark environments

Well-formedness revisited

Previous definition was enough for the semantics to coincide

equivalent queries yield the same result

while this result is meaningless

since the original SQL query is rejected by real-life RDBMS's.

Need to extend well-formedness to capture only *executable* queries.

Queries are dealing with:

1. expressions with or without aggregates,
2. formulas and
3. queries

more than 1000 lines of Coq

Real life equivalences

$$\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_1 \wedge f_2}(q).$$

```
Lemma Q_Sigma_And_Q_Sigma :
  ∀ f1 f2 q env ,
     (eval_query env (Q_Sigma f1 (Q_Sigma f2 q))) =BE= (eval_query env (Q_Sigma (
    Sql_Conj And_F f1 f2) q)).
```

⋈ is AC

```
Lemma Q_NaturalJoin_assoc :
  well_sorted_sql_table T basesort instance →
  ∀ env q1 q2 q3,
    eval_query env (Q_NaturalJoin q1 (Q_NaturalJoin q2 q3)) =BE=
    eval_query env (Q_NaturalJoin (Q_NaturalJoin q1 q2) q3).

Lemma Q_NaturalJoin_comm :
  well_sorted_sql_table T basesort instance →
  ∀ env q1 q2,
  eval_query env (Q_NaturalJoin q1 q2) =BE= eval_query env (Q_NaturalJoin q2 q1).
```

More rules ...

(5), (6) and (7) much more involved

Example

$\pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q)$ if $W_1 \subseteq W_2$

projections have to be rephrased as:

$$\pi_{\overline{e_1 \ as \ a_1}}(\pi_{\overline{e_2 \ as \ a_2}}(q)) \equiv \pi_{\overline{e_1(\overline{a_2 \leftarrow e_2}) \ as \ a_1}}(q)$$

```
Lemma Q_Pi_Flatten :
  well_sorted_sql_table T basesort instance →
  ∀ s1 s2 q env,
    let ss2 := match s2 with _Select_List s2 ⇒ s2 end in
    let f x := apply_subst_a (extract_subst ss2) x in
    let s := _Select_List
               (match s1 with
                  (_Select_List s1) ⇒
                  map (fun x ⇒ match x with
                                 | Select_As e1 a1 ⇒ Select_As (f e1) a1
                                 end) s1
                end) in
    well_formed_e T env = true →
    well_formed_q basesort env (Q_Pi s1 (Q_Pi s2 q)) = true →
    eval_query env (Q_Pi s1 (Q_Pi s2 q)) =BE= eval_query env (Q_Pi s q).
```

And more

```
Lemma Q_Sigma_Q_Pi :
  well_sorted_sql_table T basesort instance →
  ∀f s ss q env,
    extract_subst s = Some ss →
  well_formed_e T env = true →
    well_formed_q basesort env (Q_Sigma f (Q_Pi (_Select_List s) q)) = true →
    (eval_query env (Q_Sigma f (Q_Pi (_Select_List s) q))) =BE=
    (eval_query env (Q_Pi (_Select_List s) (Q_Sigma (apply_subst_frm ss f) q))).
```

```
Lemma Sigma_Join_Descend :
    well_sorted_sql_table T basesort instance →  ∀f1 q1 q2 env, well_formed_e T env
     = true →
    well_formed_q basesort env ((Q_Sigma f1 (Q_NaturalJoin q1 q2))) = true →
    (sort q2 interS attributes_sql_f (free_variables_q basesort) f1) subS sort q1 →
    (eval_query env (Q_Sigma f1 (Q_NaturalJoin q1 q2))) =BE= (eval_query env (
     Q_NaturalJoin (Q_Sigma f1 q1) q2)).
```

Optimisation verified

Coq $\overset{\text{rew}}{\text{DB}}$ as a rewriting system

Normal forms

$$\pi_{s_1 ++ \cdots ++ s_n}(\sigma_{f_1 \wedge \cdots \wedge f_m}(q_1 \bowtie \ \ldots \ \bowtie q_p))$$

Theorem (Normalization preserves semantics)

Let l be a list of Optim *and* $q \in SQL_{Alg}$. *Then:*

$$\forall WF(\mathcal{E}), \mathbb{W}^Q(q) \implies [\![normalize(q)]\!]^Q_{\mathcal{E}} = [\![q]\!]^Q_{\mathcal{E}}$$

Coq tactics

```
Parse_sql "select mid, title from movie where mid < 2000;" q.
Postgres_qep "select mid, title from movie where mid < 2000;" qep.

Goal is_valid_qep basesort_movie q qep.
Proof.
  validate_qep optims.
Qed.
```

```
Parse_sql "select m.mid, title from movie m, role r where r.mid = m.mid and year >
     1980;" q.
Postgres_qep "select m.mid, title from movie m, role r where r.mid = m.mid and year >
     1980;" qep.

Goal is_valid_qep basesort_movie q qep.
Proof.
  validate_qep optims.
```

$$[\![\text{normalize}(q)]\!]^{Q}_{\emptyset} = [\![\text{normalize}(qep)]\!]^{Q}_{\emptyset}$$

Conclusion

Coq internalisation of SQL's syntax and semantics

$\leadsto$ formal executable mechanised semantics

Coq formalisation of an extended relational algebra: SQL_{Alg}

then

Formally proved $SQL_{Coq} \equiv SQL_{Alg}$

$\leadsto$ certified logical optimisation

$\leadsto$ compellingly close a 30-year open question

Conclusion

Coq specification and implementation of SQL's engines building blocks: physical algebra

Compiler:

Verify produced strategy is semantically correct

Skeptical Approach based on traces

Rewriting

Perspectives

SQL:

`order by`, `windows`, `rank`,
regular expressions for strings (`like`)
more types : `date`

Physical algebra:

sort-based operators: `Sort scan`, `Sort-merge`

accumulators: `Aggregate`, `Hash aggregate`

nesting/correlation: `Subplan`