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
Motivations

Data are pervasive and valuable ...

Stored into relational database systems ... most widespread ones

Mature implementations

Oracle, DB2 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 → AST

Semantic analysis

Textbooks \( \text{AST}_{sem} \equiv \text{relational algebra expression} \)

Real life

AST \( \text{sem} \) depends on DB vendors

Optimisation / Query planning

AST \( \text{sem} \) → AST \( \text{sem} \) → AST \( \text{phys} \)

Logical

AST \( \text{sem} \) → AST \( \text{sem} \)

rewritings / algebraic equivalences

Physical

AST \( \text{sem} \) → AST \( \text{phys} \)

auxiliary data structures (B trees, Hash tables etc)

physical algebra – different implementations of operators

data dependent statistics
SQL’s compilation: example

\[
\pi_{\text{lastname}}(\sigma_{\text{year} > 1950}(\text{people} \bowtie \text{director} \bowtie \text{role} \bowtie \text{movie}))
\]

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;
**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;
)

\[ \pi_{\text{lastname}}(\sigma_{\text{year}>1950}(\text{people} \bowtie \text{director} \bowtie \text{role} \bowtie \text{movie})) \]
SQL's compilation: example

```sql
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;
)
```

```
πlastname(σyear>1950(people △◁ director △◁ role △◁ 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

\[ \sigma_{f_1 \land f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) \]
\[ \sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 \]

Coq Rew DB - theorems library

\[ e_{alg} = e_{qep} \]

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

validity:
check \( e_{alg} e_{qep} = true \)
\[ \Rightarrow \forall \text{db. } \text{eval db } e_{alg} = \text{eval db } e_{qep} \]
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)

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, t1(b) = 2

t2(a) = 3, t2(b) = 2

t3(a) = 1, t3(b) = 1

extract information through query languages

relational algebra: \lambda\text{-calculus for databases}
Relational model

model information

\( r(a, b) \)

through relations

schema

\( r \) relation’s name, \( a, b \) attributes, \( \{ a, b \} \) sort
denote finite sets of tuples

position vs name

\[
\begin{align*}
\text{r} &= \{(1, 2); (3, 2); (1, 1)\} \\
\text{r} &= \{t1; t2; t3\} \\
t1.a &= 1, \quad t1.b = 2 \\
t2.a &= 3, \quad t2.b = 2 \\
t3.a &= 1, \quad t3.b = 1
\end{align*}
\]

extract information

through query languages

relational algebra: \( \lambda \)-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

\[ [\sigma_f(q)] = \{ t \in [q] | [f](t) \} \]
\[ [\pi_W(q)] = \{ t|_W | t \in [q] \} \]
\[ [\rho_g(q)] = \{ t' | \exists t \in [q], \forall a \in \text{sort}(q), t'.g(a) = t.a \} \]
\[ [q_1 \bowtie q_2] = \{ t | \exists t_1 \in [q_1], \exists t_2 \in [q_2], t|_{\text{sort}(q_1)} = t_1 \land t|_{\text{sort}(q_2)} = t_2 \} \]

\[ \text{sort}(q_1) \cap \text{sort}(q_2) = \emptyset \]
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` \(\sigma\)
Builds groups with `group by` expression \(\?\)
Discards groups not satisfying `having` \(\?\)
Evaluates `select` expressions \(\pi + \rho\)
SQL : simple and declarative

\[ r_1 = \{ (a = 1, b = 4, c = 1); (a = 1, b = 5, c = 1); (a = 7, b = 4, c = 3); (a = 7, b = 1, c = 1) \} \]
\[ r_2 = \{ (c = 1, d = 4); (c = 7, d = 4) \} \]

\[
\text{select } * \text{ from } r_1, r_2 \text{ where } b \succ d ; \quad \text{position} \\
\sigma_{b \succ d}(r_1 \times r_2) \\
\neq \sigma_{b \succ d}(r_1 \bowtie r_2)
\]
\[ \{ (a = 1, b = 5, c = 1, c = 1, d = 4); (a = 1, b = 5, c = 1, c = 7, d = 4) \} \]

\[
\text{select } a \text{ from } r_1 \text{ where exists (select } d \text{ from } r_2 \text{ where } a < d) ; \\
\text{exists: test for non emptiness (}\emptyset) \]
\[ \text{nested, correlated query} \]
\[ \{ (a = 1); (a = 1) \} \]

\[ \pi_\{a\}(\sigma_{a < d}(r_1 \bowtie r_2)) \]
SQL: simple and declarative

\[ r_1 = \{(a = 1, b = 4, c = 1); (a = 1, b = 5, c = 1); (a = 7, b = 4, c = 3); (a = 7, b = 1, c = 1)\} \]

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

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
1 & 4 & 1 \\
7 & 1 & 1 \\
\end{array}
\]

select b, sum(a) from r1 where b \neq 5 group by b;

\[
\{ (a = 1, b = 4, c = 1); (a = 7, b = 4, c = 3); (a = 7, b = 1, c = 1) \}
\]

\[
\begin{array}{c|c|c}
\text{b} & \text{sum(a)} \\
4 & 8 \\
1 & 7 \\
\end{array}
\]

\[
\lambda_b, \text{sum}(a) (\sigma_{b \neq 5}(r_1))
\]
SQL: simple and declarative

```sql
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

⇒ **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: $\text{SQL}_{\text{Coq}}$
4. Define a Coq mechanised bag algebra: $\text{SQL}_{\text{Alg}}$
5. Prove the equivalence $\text{SQL}_{\text{Coq}} \equiv \text{SQL}_{\text{Alg}}$
Mechanised semantics
**SQL: NULL’s**

**NULL absorbing element in expressions**

**NULL uncomparable with anything else**

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

\[ \text{NULL} + 2 \leadsto \text{NULL} \]

\[ (\text{NULL} = \text{NULL}) \leadsto \bot \]

\[ (\text{NULL} \neq \text{NULL}) \leadsto \bot \]

\[ (\text{NULL} \neq 1) \leadsto \bot \]
SQL: aggregates, nesting, correlation

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

\[ r_2 = \{ (a_2 = 7, b_2 = 7), (a_2 = 7, b_2 = 7) \} \]

\[ Q(k): \text{ select } a_1 \text{ from } r_1 \text{ group by } a_1 \text{ having exists (select } a_2 \text{ from } r_2 \text{ group by } a_2 \text{ having expression with aggregates } = k) ; \]

How to evaluate expression? On which groups?

4 \( a_1 \)-groups for \( r_1 \)

\[ G_1 = \{ (a_1 = 1, b_1 = i) \mid 1 \leq i \leq 10 \} \quad \text{cardinality} = 10 \]
\[ G_2 = \{ (a_1 = 2, b_1 = i) \mid 1 \leq i \leq 10 \} \quad \text{cardinality} = 10 \]
\[ G_3 = \{ (a_1 = 3, b_1 = i) \mid 1 \leq i \leq 5 \} \quad \text{cardinality} = 5 \]
\[ G_4 = \{ (a_1 = 4, b_1 = i) \mid 6 \leq i \leq 10 \} \quad \text{cardinality} = 5 \]

a single \( a_2 \)-group for \( r_2 \)

\[ G' = \{ (a_2 = 7, b_2 = 7), (a_2 = 7, b_2 = 7) \} \quad \text{cardinality} = 2 \]
SQL: aggregates, nesting, correlation

$Q_1(k)$: expression $= \text{sum}(1+0*b2)$, 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

$\leadsto$ $G'$, $a_2$-group of $r_2$
SQL: aggregates, nesting, correlation

\[ Q_2(k): \text{expression} = \text{sum}(1), \text{again, computes group's cardinality} \]

\[ k = 2 \leadsto \{(a_1 = 1); (a_1 = 2); (a_1 = 3); (a_1 = 4)\} \]

\[ k \neq 2 \leadsto \{\} \]

groups with cardinality 2 \leadsto \mathcal{G}', \text{a}_2\text{-group of } r_2
SQL: aggregates, nesting, correlation

$Q_2(k)$: expression $= \text{sum}(1)$, 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 $\leadsto G'$, $a_2$-group of $r_2$

Tentative conclusion: $1+0*b2 = 1$
SQL: aggregates, nesting, correlation

\[ Q_3(k): \text{expression} = \text{sum}(1+0*b1) \]

\[ k = 5 \mapsto \{(a_1 = 3); (a_1 = 4)\} \quad k = 10 \mapsto \{(a_1 = 1); (a_1 = 2)\} \]

\[ k \neq 5 \land k \neq 10 \mapsto \{\} \]

groups with cardinality 5 and 10 \[ \mapsto \mathcal{G}_i, \ a_1\text{-group of } r_1 \]
SQL: aggregates, nesting, correlation

$Q_3(k)$: expression $= \text{sum}(1+0*b1)$

$k = 5 \mapsto \{(a_1 = 3); (a_1 = 4)\}$ \hspace{1cm} $k = 10 \mapsto \{(a_1 = 1); (a_1 = 2)\}$

$k \neq 5 \land k \neq 10 \mapsto \{\} \qquad \text{groups with cardinality 5 and 10}$

$\mapsto \quad \mathcal{G}_i, a_1$-group of $r_1$

Tentative conclusion: $1+0*b2 = 1 \not\equiv 1+0*b1$
SQL: aggregates, nesting, correlation

\[ Q_4(k): \text{expression} = \text{sum}(1+0*b1) + \text{sum}(1+0*b2) \]

\[ k = 7 \leadsto \{(a_1 = 3); (a_1 = 4)\} \quad k = 12 \leadsto \{(a_1 = 1); (a_1 = 2)\} \]

\[ k \neq 7, k \neq 12 \leadsto \{\} \]

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 $= \text{sum}(1+1*a_1+0*b_2)$

\[
    k = 4 \rightsquigarrow \{ (a_1 = 1) \} \quad k = 6 \rightsquigarrow \{ (a_1 = 2) \} \\
    k = 8 \rightsquigarrow \{ (a_1 = 3) \} \quad k = 10 \rightsquigarrow \{ (a_1 = 4) \} \\
    k \neq 4 \land k \neq 6 \land k \neq 8 \land k \neq 10 \rightsquigarrow \{ \} \\
\]

$\text{sum}(1+1*a_1+0*b_2)$ evaluated for each $a_2$-group combined with projection of $a_1$-group

$t_1 = \text{projection of } G_i \text{ on } a_1 \rightsquigarrow t_1 = (a_1 = i)$

$1+1*a_1+0*b_2$ is evaluated over tuples $t' \bowtie t_1$, where $t' \in G' \rightsquigarrow 1+i$

$G'$ has cardinality 2, $\rightsquigarrow \text{sum}(1+1*a_1+0*b_2) = 2*(1+i)$
SQL: aggregates, nesting, correlation

\[ Q_6(k) : \text{sum}(1+0*b_1+0*b_2) \]

ERROR: subquery uses ungrouped column "r1.b1" from outer query
LINE 1: ...sts (select a2 from r2 group by a2 having sum(1+0*b_1+0*b_2) =

\[ Q_7(k) : \text{sum}(1+0*b_1+0*a_2) \]

ERROR: subquery uses ungrouped column "r1.b1" from outer query
LINE 1: ...sts (select a2 from r2 group by a2 having sum(1+0*b_1+0*a_2) =
Environments: recap

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

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

\( S = (A, g, G) = \) (attributes, grouping expressions, groups of tuples)
Evaluation: recap

- **simple expression** $\leadsto$ (unique) binding for each attribute
- **function**($\overline{e}$) $\leadsto$ evaluate independently each $e_i$ of ($\overline{e}$)
- **aggregate**(cst) $\leadsto$ use innermost slice (cardinality)
- **aggregate**(e) in $[S_n; \ldots; S_1]$ $\leadsto$ 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

$S_n$ $S_{n-1}$ $S_1$

$G_n$ $G_k$ $t_{k-1}$ $t_1$
Evaluation: recap

- Simple expression \( \leadsto \) (unique) binding for each attribute
  - function(\( \bar{e} \)) \( \leadsto \) evaluate independently each \( e_i \) of \( \bar{e} \)
  - aggregate(cst) \( \leadsto \) use innermost slice (cardinality)
  - aggregate(e) \( \leadsto \) find the smallest suitable suffix \([S_k; S_{k-1}; \ldots; S_1] \)

\( \leadsto \) 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 \( k \)th slice
SQL\textsubscript{Coq} Syntax

Queries

Inductive \texttt{set\_op} := Union \mid Intersect \mid Except.
Inductive \texttt{select} := Select\_As : aggterm \to attribute \to select.
Inductive \texttt{select\_item} := Select\_Star \mid Select\_List : list select \to select\_item.
Inductive \texttt{group\_by} := Finest\_P \mid Group\_By : list funterm \to group\_by.

Inductive \texttt{att\_renaming} := Att\_As : attribute \to attribute \to att\_renaming.
Inductive \texttt{att\_renaming\_item} :=
  Att\_Ren\_Star \mid Att\_Ren\_List : list att\_renaming \to att\_renaming\_item.

Inductive \texttt{sql\_query} :=
| Table : relname \to sql\_query
| Set : set\_op \to sql\_query \to sql\_query \to sql\_query
| Select : (** select *) select\_item \to
  (** from *) list from\_item \to
  (** where *) formula sql\_query \to
  (** group by *) group\_by \to
  (** having *) formula sql\_query \to sql\_query

\textbf{with} \texttt{from\_item} := From\_Item : sql\_query \to att\_renaming\_item \to sql\_from\_item.

\textbf{where, group by and having} mandatory in SQL\textsubscript{Coq}
\textbf{no where in SQL} \leadsto TTrue
\textbf{no group by but having} in SQL \leadsto Group\_By nil
\textbf{no group by nor having} in SQL \leadsto Finest\_P \oplus TTrue
**SQL\textsubscript{Coq} Syntax**

**Formulas**

\begin{verbatim}
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.
\end{verbatim}

**FO + in + exists**

\[ \forall \rightarrow \text{all} ; \exists \rightarrow \text{any} \]

\text{in (membership)} \rightarrow _\in_ (not a usual predicate over values)

\text{exists} \rightarrow \text{non emptyness}

parameterised by \text{dom}

\text{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 \(\exists x, x \in sq \land x = \text{projection env s}\)

⚠️ In presence of \textbf{NULL} in \(x\) or in \text{projection env s} \(\rightsquigarrow \text{unknown}\)
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\textsubscript{Coq} with relational algebra

Define SQL\textsubscript{Alg} an extended relational algebra

Enjoying a bag semantics and

Natively accounting for \texttt{group by having}
**SQL\textsubscript{Alg} syntax**

Inductive \texttt{alg\_query} : Type :=
\begin{itemize}
\item \texttt{Q\_Empty\_Tuple} : \texttt{alg\_query}
\item \texttt{Q\_Table} : \texttt{relname} \to \texttt{alg\_query}
\item \texttt{Q\_Set} : \texttt{set\_op} \to \texttt{alg\_query} \to \texttt{alg\_query} \to \texttt{alg\_query}
\item \texttt{Q\_Join} : \texttt{alg\_query} \to \texttt{alg\_query} \to \texttt{alg\_query}
\item \texttt{Q\_Pi} : \texttt{list \ select} \to \texttt{alg\_query} \to \texttt{alg\_query}
\item \texttt{Q\_Sigma} : (\texttt{formula \ alg\_query}) \to \texttt{alg\_query} \to \texttt{alg\_query}
\end{itemize}

(* extending the usual $\gamma$ textbook operator *)
\begin{itemize}
\item \texttt{Q\_Gamma} :
\begin{itemize}
\item (* aggregated (output) expressions *) \texttt{list \ select} \to
\item (* grouping expressions *) \texttt{list \ funterm} \to
\item (* handling having condition *) (\texttt{formula \ alg\_query}) \to
\item (* query *) \texttt{alg\_query} \to \texttt{alg\_query}.
\end{itemize}
\end{itemize}

traditional algebra + $\gamma$ operator: \texttt{Q\_Gamma}

extending the one in to account for having

formulas shared with SQL\textsubscript{Coq}
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\textsubscript{Coq} \equiv SQL\textsubscript{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\textsubscript{Coq} query \(sq\) is well-formed when all attributes in its \texttt{from} clause are pairwise distinct (and recursively).

**Theorem**

Let \([\_]_{db}\) be well-sorted.

Let \(sq\) be a well-formed SQL\textsubscript{Coq} query, then:

\[
\forall env, \quad [T^q(sq)]^Q_{env} = [sq]^q_{env}
\]

Let \(aq\) be a SQL\textsubscript{Alg} query then:

\[
\forall env, \quad [T^Q(aq)]^q_{env} = [aq]^Q_{env}
\]
Physical algebra
### High-level specification

#### Iterator interface operators

<table>
<thead>
<tr>
<th>data centric operators</th>
<th>SQL algebra</th>
<th>$\phi$ algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>index based</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simple</td>
</tr>
<tr>
<td>map</td>
<td>$r, \pi$</td>
<td>Seq Scan</td>
</tr>
<tr>
<td>join</td>
<td>$\times$</td>
<td>Nested loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block nested loop</td>
</tr>
<tr>
<td>filter</td>
<td>$\sigma$</td>
<td>Filter</td>
</tr>
<tr>
<td>group</td>
<td>$\gamma$</td>
<td>Group</td>
</tr>
</tbody>
</table>

#### Intermediate results storage operators

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Materialize</td>
</tr>
</tbody>
</table>

- map
- join
- filter
- group

iterate  
combine  
extract according to a condition  
aggregate results

**specify these operations**  **high degree of abstraction**
Main idea

High-Level Spec

Definition \text{is_a...op p o} :=
\forall x t, \text{nb_occ t} (o p x) = f_{o,p}(t, \text{nb_occ t x})

\phi\text{-algebra}

Lemma \phi\_...\_op\_is\_a\_...\_op :
H_\phi \Rightarrow \forall x t, \text{nb_occ t} (o_\phi p x) = f_{o,p}(t, \text{nb_occ t x})

SQL Algebra

Lemma SQL\_...\_op\_is\_a\_...\_op :
H_{SQL} \Rightarrow \forall x t, \text{nb_occ t} (o_{SQL} p x) = f_{o,p}(t, \text{nb_occ t x})

Bridge

Lemma \phi\_...\_op\_implements\_SQL\_...\_op :
H_\phi \wedge H_{SQL} \Rightarrow \forall x t, \text{nb_occ t} (o_\phi p x) = \text{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
\textit{QEP}_\text{Coq} \text{ a domain specific language for QEP's}
QEP\textsubscript{Coq} : A QEP language

cursor ::=  
  | SeqScan \(table\)  
  | Projection \((e^a \text{ as attribute})\) cursor  
  | Filter formula cursor  
  | NestedLoop cursor cursor  
  | ThetaNestedLoop formula cursor cursor  
  | Group \(e^f\) formula \((e^a \text{ as attribute})\) cursor  
  | IndexScan index value  
  | IndexJoin \(e^f\) cursor index

index ::=  
  | FilterIndex \(table \ e^f \ p \quad p \in \text{predicate}\)  
  | HashIndex cursor \(e^f\)
Theorem \((\text{QEP}_{\text{Coq}} \equiv \text{SQL}_{\text{Alg}})\)

Given a well-sorted database instance and \(q\) a \(\text{QEP}_{\text{Coq}}\), then:

\[
\mathcal{W}^{\text{QEP}}(q) \implies [T^{\text{QEP}}(q)]^Q_{\emptyset} = [q]^{\text{QEP}}
\]

\(\mathcal{W}^{\text{QEP}}()\) well formed condition for QEP’s

similar to well-formedness for SQL\(_{\text{Alg}}\) queries

\(T^{\text{QEP}}(q)\)

translation from QEP’s to SQL\(_{\text{Alg}}\) expressions
Coq\textsubscript{ rew\_DB}: a DB rewritings library
Textbooks’ equivalences

\[ \sigma_{f_1 \land f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) \quad \text{(1)} \]
\[ \sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q)) \quad \text{(2)} \]
\[ (q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3) \quad \text{(3)} \]
\[ q_1 \bowtie q_2 \equiv q_2 \bowtie q_1 \quad \text{(4)} \]

\[ \pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) \quad \text{if } W_1 \subseteq W_2 \quad \text{(5)} \]
\[ \pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q)) \quad \text{if } \text{Att}(f) \subseteq W \quad \text{(6)} \]
\[ \sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 \quad \text{if } \text{Att}(f) \subseteq \text{sort}(q_1) \quad \text{(7)} \]
\[ \sigma_f(q_1 \mathord{\Box} q_2) \equiv \sigma_f(q_1) \mathord{\Box} \sigma_f(q_2) \quad \text{where } \mathord{\Box} \text{ is } \cup \text{ or } \cap \quad \text{(8)} \]
Caveat

relational algebra projections

\[ \pi_W \]

\[ W, \text{ a finite set of attributes} \]

SQL more complex as projections (select clause)

involve complex expressions and substitutions

\[ \pi e \text{ as } a. \]

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

\[ \sigma_{f_1} (\sigma_{f_2} (q)) \equiv \sigma_{f_1 \land f_2} (q). \]

Lemma Q_Sigma_And_Q_Sigma :
\[ \forall f_1 \ f_2 \ q \ \text{env}, \]
\[ (\text{eval_query env (Q_Sigma f_1 (Q_Sigma f_2 q))) \ = \ BE = \ (\text{eval_query env (Q_Sigma (Sql_Conj And_F f_1 f_2) q))}). \]
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

\[ \pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) \text{ if } W_1 \subseteq W_2 \]

projections have to be rephrased as:

\[ \pi_{e_1} as_{a_1}(\pi_{e_2} as_{a_2}(q)) \equiv \pi_{e_1(a_2 \leftarrow e_2)} as_{a_1}(q) \]

Lemma Q_Pi_Flatten :

\text{well_sorted_sql_table T basesort instance} \rightarrow
\forall s_1 \ s_2 \ q \ env, \hspace{1cm}
\begin{align*}
\text{let ss2 := match } s_2 \text{ with } _\text{Select_List} s_2 \Rightarrow s_2 \text{ end in} \\
\text{let } f \ x := \text{apply_subst_a (extract_subst ss2)} \ x \ \text{in} \\
\text{let } s := _\text{Select_List} \left( \text{match } s_1 \text{ with} \\
( _\text{Select_List} s_1) \Rightarrow \text{map (fun } x \Rightarrow \text{match } x \text{ with} \\
| \text{Select_As } e_1 a_1 \Rightarrow \text{Select_As}(f \ e_1) a_1 \\
\text{end) } s_1 \right) \\
\text{well_formed_e T env = true } \rightarrow \\
\text{well_formed_q basesort env (Q_Pi s_1 (Q_Pi s_2 q)) = true } \rightarrow \\
\text{eval_query env (Q_Pi s_1 (Q_Pi s_2 q)) = BE = eval_query env (Q_Pi s q).}
\end{align*}
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\textsubscript{rew\_DB} as a rewriting system

Normal forms

\[ \pi_{s_1\ldots s_n}(\sigma_{f_1\land\ldots\land f_m}(q_1\bowtie\ldots\bowtie q_p)) \]

Theorem (Normalization preserves semantics)

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

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

\[ \llbracket \text{normalize}(q) \rrbracket_{\emptyset}^Q = \llbracket \text{normalize}(qep) \rrbracket_{\emptyset}^Q \]
Conclusion

Coq internalisation of SQL’s syntax and semantics

\[ \leadsto \text{formal executable mechanised semantics} \]

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

then

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

\[ \leadsto \text{certified logical optimisation} \]

\[ \leadsto \text{compellingly close a 30-year open question} \]
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