A Coq formalisation of SQL's execution engines*

V. Benzaken[2], É. Contejean[1], Ch. Keller[2], and E. Martins[2]

[1] CNRS, Université Paris Sud, LRI, France
[2] Université Paris Sud, LRI, France

Abstract. In this article, we use the Coq proof assistant to *specify* and *verify* the low level layer of SQL's execution engines. To reach our goals, we first design a high-level Coq specification for data-centric operators intended to capture their essence. We, then, provide two Coq implementations of our specification. The first one, the physical algebra, consists in the low level operators found in systems such as Postgresql or Oracle. The second, SQL algebra, is an extended relational algebra that provides a semantics for SQL. Last, we formally relate physical algebra and SQL algebra. By proving that the physical algebra implements SQL algebra, we give high level assurances that physical algebraic and SQL algebra expressions enjoy the same semantics. All this yields the first, to our best knowledge, formalisation and verification of the *low level layer of an RDBMS* as well as SQL's compilation's *physical optimisation*: fundamental steps towards mechanising SQL's compilation chain.

1 Introduction

Data-centric applications involve increasingly massive data volumes. An important part of such data is handled by relational database management systems (RDBMS's) through the SQL query language. Surprisingly, formal methods have not been broadly promoted for data-centric systems to ensure strong *safety* guarantees about their expected behaviours. Such guarantees can be obtained by using proof assistants like Coq [27] or Isabelle [28] for specifying, proving and testing (parts of) such systems. In this article, we use the Coq proof assistant to *specify* and *verify* the low level layer of an RDBMS as proposed in [26] and detailed in [18].

The theoretical foundations for RDBMS's go back to the 70's where relational algebra was originally defined by Codd [13]. Few years later, SQL, the standard domain specific language for manipulating relational data was designed [10]. SQL was dedicated to *efficiently* retrieve data stored on *secondary storage* in RDBMS's, as described in the seminal work [26] that addressed the low level layer as well as secondary memory access for such systems, known in the field as *physical algebra*, *access methods* and *iterator interface*. SQL and RDBMS's evolved over time but they still obey the principles described in those works and found in all textbooks on the topic (see [18,25,15,5] for instance). In particular, the *semantic analysis* of a SQL query could yield an expression, e_1, of

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

an (extended) relational algebra. The *logical* optimisation step rewrites this expression into another algebraic expression e_2 (based on well-known rules that can be found in [18]). Then *physical* optimisation takes place and an evaluation strategy for expression e_2, called a *query execution plan (QEP)* in this setting, is produced. QEP's are composed by *physical algebra operators*. Yet there are no formal guarantees that the produced QEP and (optimised) algebraic expression do have the same semantics. One contribution of our work is to open the way to formally provide such evidences. To reach our goal, we adopt a very general approach that is not limited to our specific problem. It consists in providing a high-level pivotal specification that will be used to describe and relate several lower-level languages.

In our particular setting, we first design a high-level, very abstract, generic, thus extensible, Coq specification for data-centric operators intended to capture their essence (which will be useful to address other data models and languages than relational ones).

The first low-level language consists of physical operators as found in systems such as Postgresql and described in main textbooks on the topic [18,25]. One specificity and difficulty lied in the fact that, when evaluating a SQL query, all those operators are put together, and for efficiency purposes, database systems implement, as far as possible, on-line ([22]) versions of them through the iterator interface. At that point there is a discrepancy between the specifications that provide collection-at-a-time invariants and the implementations that account for value-at-a-time executions. To fill up the gap, we exhibit non trivial invariants to prove that our on-line algorithms do implement their high-level specification. Moreover, those operators are shown to be exhaustive and to terminate.

The second low-level language (actually mid-level specification) is SQL algebra (syntax and semantics), an algebra that hosts SQL. By hosting we mean that there is an embedding of SQL into this algebra which preserves SQL's semantics. Due to space limitations, such an embedding is out of the scope of this paper and is described in [7]. We relate each algebraic operator to our high level specification by proving adequacy lemmas providing strong guarantees that the operator at issue is a realization of the specification.

Last, we formally bridge both implementations. By proving that the physical algebra does implement SQL algebra, we give strong assurances that the QEP and the algebraic expression resulting from the semantics analysis and logical optimisation do have the same semantics. This last step has been eased thanks to the efforts devoted to the design of our high-level pivotal specification. All this yields the first, to our best knowledge, *executable* formalisation and verification of the *low level layer of an RDBMS* as well as SQL's compilation's *physical optimisation*: fundamental steps towards mechanising SQL's compilation chain.

Organisation We briefly recall in Section 2 the key ingredients of SQL compilation and database engines: extended relational algebra, physical algebra operators and iterator interface. Section 3 presents our Coq high-level specification that captures the essence of data-centric operators. In Section 4, we formalise the iterator interface and physical algebra, detailing the necessary invariants. Section 5 presents the formal specification of SQL algebra. We formally establish, in Section 6, that any given physical operator does implement its corresponding logical operator. We draw lessons, compare our work, conclude and give perspectives in Section 7.

2 SQL's compilation in a nutshell

Following [18] SQL's compilation proceeds into three broad steps. First, the query is *parsed*, that is turned into a parse-tree representing its structure. Second, *semantics analysis* is performed transforming the parse tree into an expression tree of (extended) *relational algebra*. Third, the *optimisation* step is performed: using relational algebraic rewritings (logical optimisation) and based on a cost model [3], a physical *query execution plan (QEP)* is produced. It not only indicates the operations performed but also the order in which they will be evaluated, the algorithm chosen for each operation and the way stored data is obtained and passed from one operation to another. This last stage is *data dependent*.

We present the main concepts through the following example that models a movie database gathering information about movies (relation `movie`), movies' directors `director`, the movies they directed and relation `role` carrying information about who played (identified by his/her `pid`) which role in a given movie (identified by its `mid`). On Figure 1 we give for a typical SQL query the corresponding (Postgresql)[4] QEP issued as well as the AST obtained after semantic analysis and logical optimisation.

```
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;
```

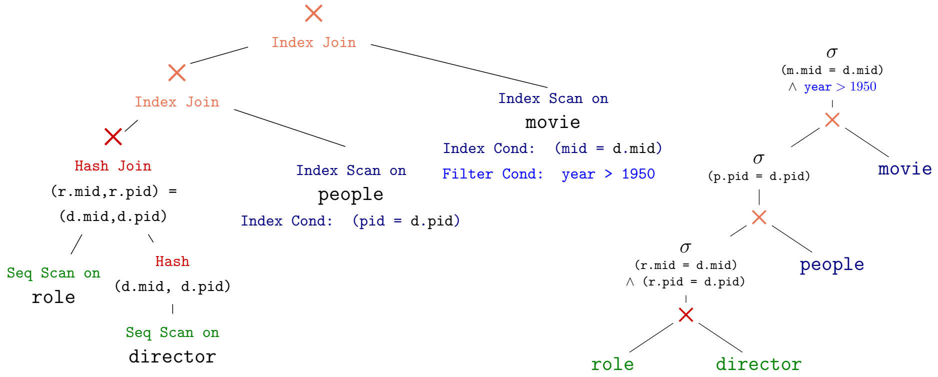


Fig. 1: A typical SQL query, its QEP and logical AST.

[3] The model exploits system collected statistics about the data stored in the database.

[4] The `IJ` nodes are expressed in Postqresql as `Nested loop` combined with an Index scan but corresponds to an index-based join.

The leaves (*i.e.*, relations) are treated by means of access methods such as `Seq Scan` or `Index Scan` (in case an index is available); a third access method usually provided by RDBMS's is the `Sort Scan` which orders the elements in the result according to a given criteria. In the example, relations `role` and `director` are accessed via `Seq Scan`, whereas `people` and `movie` are accessed thanks to `Index Scan`. The product of relations in the `from` part is reordered and the filtering condition is spread over the relevant (sub-product of) relations.

Intuitively, each physical operator corresponds to one or a combination of algebraic operators: σ (selection), $\times$ (product), completed with π (projection) and γ (grouping) (see Section 5.1 for their formal semantics).

Conversely, to each operator of the logical plan, $\sigma, \times, \ldots$, potentially corresponds one or more operators of the physical plan: the underlying database system provides several different algorithm's implementations. For the cross product, for instance, at least four such different algorithms are provided by mainstream systems: `Nested Loop`, `Index Join`, `Sort Merge Join` and `Hash Join`. For the selection operator the system may use the `Filter` physical operator.

The situation is made even more complex by the facts that a QEP contains some strategy (top-down, left-most evaluation) and that some physical operators are implemented via on-line algorithms. Hence a filtering condition which spans over a cross-product between two operands, in an algebraic expression, may be used in the corresponding QEP to filter the second one, by inlining the condition for each tuple of the first operand. This is the case for instance with the second join of Figure 1 where the second operand is an Index-Scan. Therefore the pattern $x \times_{\texttt{IJ}}$ (`Index Scan` y `Index Cond :`$a = x.a'$) corresponds to $\sigma_{y.a=x.a'}(x \times y)$.

Unfortunately not all physical operators support the on-line approach and *materialising* partial results (*i.e.*, temporarily storing intermediate results) is needed: the `Materialise` physical operator allows to express this in Postgresql physical plans. Table 1 summarises our contributions where the colored cells indicate the Coq specified and implemented operators.

Iterator interface operators				
Section 3 data centric operators	**Section 4, ϕ algebra**			**Section 5 SQL algebra**
	simple	**index based**	**sort based**	

map	`Seq Scan`	`Index scan`
		`Bitmap index scan`
			`Sort scan`	r, π

join	`Nested loop`
	`Block nested loop`
		`Hash join`
		`Index join`
			`Sort merge join`	$\times$

filter	`Filter`			σ
group	`Group`			γ
bind	`Subplan`			`env`
accumulator	`Aggregate`, `Hash`, `Hash aggregate`			aggregate

	Intermediate results storage operators `Materialize`	

Table 1: Synthesis

3 A high-level specification for data-centric operators

In the data-centric setting, data are mainly collections of values. Such values can be combined and enjoy a decidable comparison. Operators allow for manipulating collections, that is to *extract* data from a collection according to a condition (filter), to *iterate* over a collection (map), to *combine* two collections (join) and, last, to *aggregate* results over a collection (group).

Since collections may be implemented by various means (lists with or without dupplicates, AVL, etc), in the following we shall call these implementations `containerX`'s. The content, that is the elements gathered in such a `containerX`, may be retrieved with the corresponding function `contentX` and we also make a last assumption, that there is a decidable equivalence `equivX` for elements. The function `nb_occX` is defined as the the number of occurrences of an element in the `contentX` of a `containerX` modulo `equivX`[5].

We then characterise the essence of data centric operations performed on containers. Operators filter and map are a lifting of the usual operators on lists to containers.

```
Definition is_a_filter_op contentA contentA' (f: A → bool) (fltr: containerA → containerA')
  := ∀s, ∀t, nb_occA' t (fltr s) = (nb_occA t s) * (if f t then 1 else 0).
Definition is_a_map_op contentA contentB (f: A → B) (mp: containerA → containerB) :=
   ∀s, ∀t, nb_occB t (mp s) = nb_occ t (map f (contentA s)).
```

Unlike the first two operators which make no hypothesis on the nature of the elements of a `containerX`, joins manipulate *homogeneous* containers *i.e.*, their elements are equipped with a support `supX` which returns a set of attributes, and all elements in a `containerX` enjoy the same `supX`, which is called the `sort` of the container. Let us denote by `A1` (resp. `A2`, resp. `A`) the type of the elements of the first operand (resp. the second operand, resp. the result) of a join operator `j`. Elements of type `A` are also equipped with two functions `projA1` and `projA2`, which respectively project them over `A1` and `A2`.

```
Definition is_a_join_op sa1 sa2 contentA1 contentA2 contentA
                                (j : containerA1 → containerA2 → containerA) :=
  ∀s1 s2, (∀t, 0 < nb_occA1 t s1 → supA1 t = sa1) →
                (∀t, 0 < nb_occA2 t s2 → supA2 t = sa2) →
  ((∀t, 0 < nb_occA t (j s1 s2) → supA t = (sa1 unionS sa2)) ∧
  (nb_occA t (j s1 s2) = nb_occA1 (projA1 t) s1 * nb_occA2 (projA2 t) s2))
                          * (if supA t = (sa1 unionS sa2) then 1 else 0).
```

Intuitively, joins allow for combining two homogeneous containers by taking the union of their `sort` and the product of their occurrence's functions.

The grouping operator, as presented in textbooks [18], partitions, using `mk_g`, a container into groups according to a grouping criteria `g` and then discards some groups that do not satisfy a filtering condition `f`. Last for the remaining groups it `builds` a new element.

[5] `X` will be `A`, `A'`, `B`, according to the various types of elements and various implementations for the collection. A particular case of `nb_occX` is `nb_occ` which denotes the number of occurrences in a list.

```
Definition is_a_grouping_op (G : Type) (mk_g : G → containerA → list B) grp :=
 ∀(g : G) (f : B → bool) (build : B → A) (s : containerA) t,
  nb_occA t (grp g f build s) = nb_occ t (map build (filter f (mk_g g s))).
```

All the above definitions share a common pattern: they state that the number of occurences `nb_occX t (o p s)` of an element `t` in a container built from an operator `o` applied to some parameters `p` and some operands `s`, is equal to `foopp (t, nb_occX (g t) s)`, where `foopp` is a function which depends only on the operator and the parameters. This implies that any two operators satisfying the same specification `is_a_..._op` are *interchangeable*. For grouping, the situation is slightly more subtle, however the same interchangeability property shall hold since `nb_occA t (grp g f build s))` depends only on `t` and `contentA s` for the grouping criteria used in the following sections.

Tuning those definitions was really challenging: finding the relevant level of abstraction for containers and contents suitable to host both physical and logical operators was not intuitive. Even for the most simple one such as filter, we would have expected that the type of containers should be the same for input and output. It was not possible as we wanted a simple, concise and efficient implementation.

4 Physical algebra

All physical operators that can be implemented by on-line algorithms rely on a common iterator interface that allows them to build the next tuple on demand.

4.1 Iterators

A key aspect in our formalisation of physical operators is a specification of such a common iterator interface together with the properties an iterator needs to satisfy. We validate this interface by implementing standard iterative physical operators, namely sequential scanning, filtering, and nested loop.

Abstract iterator interface An iterator is a data structure that iterates over a collection of elements to provide them, on demand, one after the other. Following the iterator interface given in [18] and in the same spirit of the formalisation of cursors presented in [17], we define a `cursor` as an abstract object over some type `elt` of elements that must support three operations: `next`, that returns the next element of the iteration if it exists; `has_next`, that checks if such an element does exist; and `reset`, that restarts the cursor at its beginning. In Coq, this can be modelled as a record[6] named `Cursor` that contains (at least) an abstract type of `cursors` and these three operations:

[6] We could also use a module type, but the syntax would be heavier and less general.

```
Record Cursor (elt : Type) : Type :=
  { cursor : Type;
    next : cursor → result elt * cursor;
    has_next : cursor → Prop;
    reset : cursor → cursor;
    [...] (* Some properties, see below *) }.
```

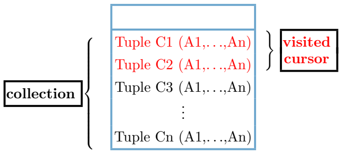
Due to the immutable nature of Coq objects, the operations `next` and `reset` must return the modified cursor. Moreover, since `next` must be a total function, a monadic[7] construction is used to wrap the element of type `t` that it outputs:

```
Inductive result (A : Type) := Result : A → result A | No_Result | Empty_Cursor.
```

The constructor `Result` corresponds to the case where an element can be returned, and the two constructors `No_Result` and `Empty_Cursor` deal with the cases where an element cannot be returned, respectively because it does not match some selection condition (see Sec. 4.1) or because the cursor has been fully iterated over.

We designed a sufficient set of properties that a cursor should satisfy in order to be valid. These properties are expressed in terms of three high-level inspection functions (that are used for specification only, not for computation): `collection` returns all the elements of the cursor, `visited` returns the elements visited so far, and `coherent` states an invariant that the given cursor must preserve:

```
Record Cursor (elt : Type) : Type := { [...]
    collection : cursor → list elt;
    visited : cursor → list elt;
    coherent : cursor → Prop; [...] }.
```



Given these operations, the required properties are the following:

```
Record Cursor (elt : Type) : Type := { [...]
 (* next preserves the collection *)
 next_col : ∀c, coherent c → collection (snd (next c)) = collection c;
 (* next adds the returned element to visited *)
 next_visited_Result :
   ∀a c c', coherent c → next c = (Result a, c') → visited c' = a :: (visited c);
 next_visited_No_Result :
   ∀c c', coherent c → next c = (No_Result, c') → visited c' = visited c;
 next_visited_Empty_Cursor :
   ∀c c', coherent c → next c = (Empty_Cursor, c') → visited c' = visited c;
 (* next preserves coherence *) next_coherent : ∀c, coherent c → coherent (snd (next c));
 (* when a cursor has no element left, visited contains all the elements of the collection *)
 has_next_spec : ∀c, coherent c → ¬ has_next c → (collection c) = (rev (visited c));
 (* a cursor has new elements if and only if next may return something *)
 has_next_next_neg : ∀c, coherent c → (has_next c ↔ fst (next c) ≠ Empty_Cursor);
 (* reset preserves the collection *)
 reset_collection : ∀c, collection (reset c) = collection c;
 (* reset restarts the visited elements *) reset_visited : ∀c, visited (reset c) = nil;
 (* reset returns a coherent cursor *) reset_coherent : ∀c, coherent (reset c); [...]}.
```

[7] This construction is similar to the exception monad. There is no interest to write the standard “return” and “bind” operators. The sequential scan and nested loop, respecitvely, can be seen as online versions of them.

The `..._coherent` and `..._collection` axioms ensure that `coherent` and the collection of elements are indeed invariants of the iterator. The `..._visited` axioms explain how visited is populated. Finally, the `has_next_spec` axiom is the key property to express that all the elements have been visited at the end of the iteration.

Last, we require a progress property on cursors (otherwise `next` could return the `No_Result` value forever and still satisfy all the properties). Progress is stated in terms of an upper bound on the number of iterations of `next` before reaching an `Empty_Cursor`:

```
Record Cursor (elt : Type) : Type := { [...]
  (* an upper bound on the number of iterations before the cursor has  been fully visited *)
  ubound : cursor → nat;
  (* this upper bound is indeed complete *)
  ubound_complete : ∀c acc, coherent c → ¬ has_next (fst (iter next (ubound c) c acc)); }.
```

where `iter f n c acc` iterates `n` times the function `f` on the cursor `c`, returning a pair of the resulting cursor and the accumulator `acc` augmented with the elements produced during the iteration. The upper bound is not only part of the specification (to state that cursors have a finite number of possible iterations) but can also be used in Coq to actually materialize them.

We will see that these properties are strong enough both to combine iterators and to derive their adequacy with respect to their algebraic counterparts.

First instance: sequential scan The base cursor implements sequential scan by returning, tuple by tuple, all the elements of a given relation, represented by a list in our high-level setting. It simply maintains a list of elements still to be visited named `to_visit` and its invariant expresses that the collection contains the elements visited so far and the elements that remain to be visited. A natural upper bound on the number of iterations is the number of elements to visit.

```
Definition coherent (c : cursor) := c.(collection) = rev c.(visited) ++ c.(to_visit).
Definition ubound (c : cursor) : nat := List.length c.(to_visit).
```

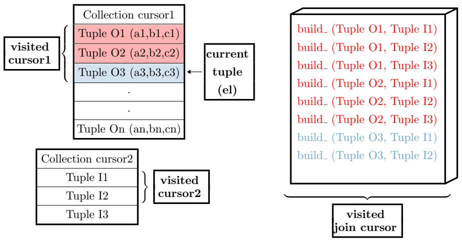
Second instance: filter Filtering a cursor returns the same cursor, but with a different function `next` and accordingly different specification functions. Given a property on the elements `f : elt → bool`, the function `next` filters elements of the underlying cursor:

```
Definition next (c : cursor) : result elt * cursor  :=
 match Cursor.next c with
  | (Result e, c’) ⇒ if f e then (Result e, c’) else (No_Result, c’) | rc’ ⇒ rc’
  end.
```

This is where `No_Result` is introduced when the condition is not met. Accordingly, the functions `collection` and `visited` are the filtered `collection` and `visited` of the underlying cursor and an upper bound on the number of iterations is the upper bound of the underlying cursor:

```
Definition collection (c : cursor) :=  List.filter f (Cursor.collection c).
Definition visited (c : cursor) := List.filter f (Cursor.visited c).
Definition ubound (q : cursor) : nat := Cursor.ubound q.
```

Third instance: nested loop The nested loop operator builds the cross-product between an outer cursor and an inner cursor: the `next` function returns either the combination of the current tuple of the outer cursor with the next tuple of the inner cursor (if this latter exists) or the combination of the next tuple of the outer cursor with the first tuple of the reset outer cursor (see Fig. 2).


Collection cursor1
Tuple O1 (a1,b1,c1)
Tuple O2 (a2,b2,c2)
Tuple O3 (a3,b3,c3)
.
.
Tuple On (an,bn,cn)
visited cursor1
current tuple (el)
Collection cursor2
Tuple I1
Tuple I2
Tuple I3
visited cursor2
build_ (Tuple O1, Tuple I1)
build_ (Tuple O1, Tuple I2)
build_ (Tuple O1, Tuple I3)
build_ (Tuple O2, Tuple I1)
build_ (Tuple O2, Tuple I2)
build_ (Tuple O2, Tuple I3)
build_ (Tuple O3, Tuple I1)
build_ (Tuple O3, Tuple I2)
visited join cursor

Algorithm 1 Nested Loop-Join

```
for each tuple o ∈ O do
    for each tuple i ∈ I do
        Add the tuple <o,i> to visited
    end for
end for
```

Fig. 2: Nested Loop-Join

Specifying such a cursor becomes slightly more involved. For correctness, one has to show the invariant stating that the elements visited so far contain (i) the last visited element of the outer cursor combined with all the visited elements of the inner cursor; and (ii) the other visited elements of the outer cursor combined with all the collection of the inner cursor.

```
Definition coherent (c: cursor) : Prop :=
  (* the two underlying cursors are coherent *)
  Cursor.coherent (outer c) ∧ Cursor.coherent (inner c) ∧
  match Cursor.visited (outer c) with
  (* if the outer cursor has not been visited yet, so as the inner cursor *)
  | nil ⇒ visited c = nil ∧ Cursor.visited (inner c) = nil
  (* otherwise, the visited elements are a partial cross-product *)
  | el :: li ⇒ visited c = (cross (el::nil) (Cursor.visited (inner c))) ++
                           (cross li (rev (Cursor.collection (inner c))))
  end.
```

where `cross` builds the cross product of two lists. For progress, an upper bound for the length of this partial cross-product is needed:

```
Definition ubound (c:cursor) : nat :=
   Cursor.ubound (inner c) +
   (Cursor.ubound (outer c) * (S (Cursor.ubound (Cursor.reset (inner c))))).
```

where a successor on the upper bound on the inner cursor has been added for simplicity reasons. The proof of completeness is elaborate and relies on key properties on bounds for cursors stating in particular that the bound decreases when `next` is applied to a non-empty cursor:

```
Lemma ubound_next_not_Empty:
  ∀c, coherent c → fst (next c) ≠ Empty_Cursor → ubound (snd (next c)) < ubound c;
```

Materialisation Independently from any specific operator, materialising an iterator is achieved by resetting it, then iterating the upper bound number of times while accumulating the returned elements. We can show the key lemma for adequacy of operators: materialising an iterator produces all the elements of its collection.

```
Definition materialize c :=
   let c' := reset c in List.rev (snd (iter next (ubound c') c' nil)).
Lemma materialize_collection c : materialize c = collection c.
```

We used the same technique to implement the grouping operator by, instead of simply accumulating the elements, group them on the fly.

4.2 Index-based operators

Having an index on a given relation is modelled as a wrapper around cursors: such a relation must be able to provide a (possibly empty) cursor for each value of the index. The main components of an indexed relation are: (i) a type `containers` of the internal representation of data (which can be a hash table, a B-tree, a bitmap, . . .), (ii) a function `proj`, representing the projection from tuples to their values on the attributes enjoying the index, (iii) a comparison function `P` on these attributes (which can be an equality for hash-indices, a comparison for tree-based indices, . . .) and (iv) an indexing function `i` that, given a container and an index, returns the cursors of the elements of the container matched by the index (w.r.t. `P`). This is implemented as the following record:

```
Record Index (elt eltp : Type) : Type :=
  { containers : Type; (* representation of data *)
    proj : elt → eltp; (* projection on the index *)
    P : eltp → eltp → bool; (* comparison between two indices *)
    i : containers → eltp → Cursor.cursor; (* indexing function *)  [...] }.
```

As for sequential iterators, we state the main three properties that an index should satisfy. Again, these properties are expressed in terms of the collection of a container, used for specification purposes only.

```
Record Index (elt eltp : Type) : Type := { [...]
  ccollection : containers → list elt; (* the elements of a container *)
  (* the collection of an indexed cursor contains the filtered elements of the
     container w.r.t. P *)
  i_collection : ∀c x, Cursor.collection (i c x) =
                            List.filter (fun y ⇒ P x (proj y)) (ccollection c);
  (* a fresh indexed cursor has not been visited yet *)
  i_visited : ∀c x, Cursor.visited (i c x) = nil;
  (* a fresh indexed cursor is coherent *) i_coherent : ∀c x, Cursor.coherent (i c x) }.
```

First instance: sequential scan Let us start with a simple example: sequential scan can be seen as an index scan with a trivial comparison function that always returns true, and a trivial indexing function that returns a sequential cursor. It is thus sufficient to use the following definitions and the properties follow immediately:

```
Definition containers := list elt.
Definition P := fun _ _ ⇒ true.
Definition i := fun c _ ⇒ SeqScan.mk_cursor c.
```

Let us see how this setting models more interesting index-based algorithms.

Second instance: hash-index scan In this case, the comparison function is an equality, and the underlying containers are hash tables whose keys are the attributes composing the index. To each key is associated the cursor whose collection contains elements whose projection on the index equals the key. In our development, we use the Coq `FMap` library to represent hash tables, but we are rather independent of the representation:

```
Record containers : Type := mk_containers
{ (* the hash table *) hash : FMapWeakList.Raw(Eltp) (cursor C);
  (* the elements are associated to the corresponding key *)
  keys : ∀x es, MapsTo x es hash → ∀e, List.In e (collection es) → P x (proj e) = true;
  noDup : NoDup hash (* the hash table has no key duplicate *) }.
```

where `MapsTo x es hash` means that `es` is the cursor associated to the key `x` in the hash table.

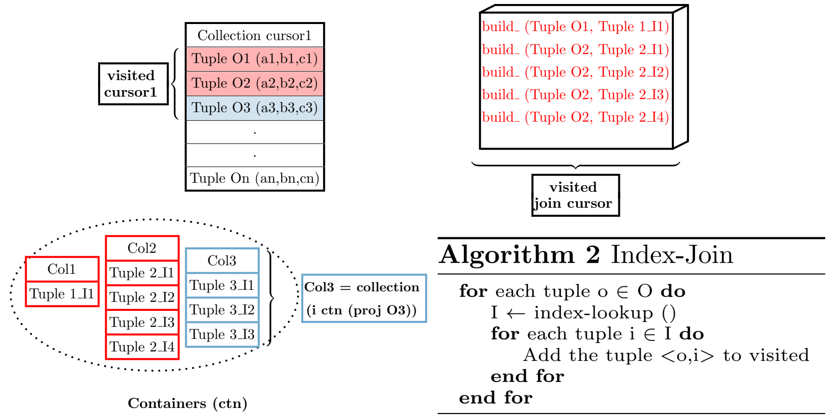
Given a particular index, the indexing function returns the cursor associated to the index in the hash table. Its properties follow from the properties of hash tables.

Third instance: bitmap-index scan In this case, the comparison function can be any predicate, and the containers are arrays of all the possible elements of the relation together with bitmaps (bit vectors) associated to each index, stating whether the n^{th} element of the relation corresponds to the index. In our development, we use Coq vectors to represent this data structure:

```
Record containers : Type := mk_containers
{ size : nat; (* the number of elements in the relation *)
  collection : Vector.t elt size; (* all the elements of the relation *)
  bitmap : eltp → Bvector size;(* a bitmap associated to every index *)
  (* each bitmap associates to true exactly the elements matching the corresponding index *)
  coherent : ∀n x0, nth (bitmap x0) n = P x0 (proj (nth collection n)) }.
```

Given a particular index, the indexing function returns the sequential cursor built from the elements for which the bitmap associated to the index returns true. Its properties follow by induction on the size of the relation.

Application: Index-join algorithm The index-join algorithm is similar in principle to the nested loop algorithm but faster thanks to an exploitable index on the inner relation: for each tuple of the outer relation, only matching tuples of the inner relation are considered (see Fig. 3). Hence, our formal development is similar as the one for nested loop, but more involved: (i) in the function `next`, each time we get a new element from the outer relation, we need to generate the cursor corresponding to the index from the inner relation (instead of resetting the whole cursor) (ii) the `collection` is now a *dependent* cross-product between the outer relation and the matching inner tuples; the invariant predicate `coherent` has to be changed consequently (iii) the `ubound` is a *dependent* product of the bound of the outer relation with each bound of the matching cursors of the inner relation (obtained by materialising the outer relation).



Algorithm 2 Index-Join

```
for each tuple o ∈ O do
    I ← index-lookup ()
    for each tuple i ∈ I do
        Add the tuple <o,i> to visited
    end for
end for
```

Fig. 3: Index-based nested loop

Derived operators Our high level of abstraction gives for free the specification of common variants of the physical operators. For instance, the Block Nested Loop algorithm is straightforwardly formalised by replacing, in the Nested Loop formalisation, the abstract type of elements by a type of "blocks" of elements (e.g., lists), and the function that combines two tuples by a function that combines two blocks of tuples.

4.3 Adequacy

All physical operators specified and implemented so far are shown to fulfil the high-level specification. For instance, if `C` is a cursor, `f` a filtering condition compatible with the equivalence of elements in `C`, then the corresponding filter iterator F:= `(Filter.build f f_eq C)` fulfils the specification of a filter:

```
Lemma mk_filter_is_a_filter_op :
  is_a_filter_op (Cursor.materialize C) (Cursor.materialize F) f (Filter.mk_filter F).
```

Sometimes, there are some additional side conditions: if `C1` and `C2` are two cursors, and `NL` := `(NestedLoop.build [...] C1 C2)` is the corresponding nested loop which combines elements thanks to the combination function `build_`, not only some hypotheses are needed to be able to build `NL`, but some extra ones are needed to prove `NL` is indeed a join operator:

```
Hypothesis [...]
Hypothesis build_split_eq_1 :
  ∀t1 u1 t2 u2, equivA (build_ t1 t2) (build_ u1 u2) → [...] → equivA1 t1 u1.
Hypothesis build_split_eq_2 :
  ∀t1 u1 t2 u2, equivA (build_ t1 t2) (build_ u1 u2) → [...] → equivA2 t2 u2.
Lemma NL_is_a_join_op :
  is_a_join_op [...] (Cursor.materialize C1) (Cursor.materialize C2) (Cursor.materialize NL)
          [...] (fun c1 c2 ⇒ NestedLoop.mk_cursor C1 C2 nil c1 c2).
```

5 SQL algebra

We now present SQL algebra, our Coq formalisation of an algebra that satisfies the high-level specification given in Section 3 and that hosts SQL.

5.1 Syntax and semantics

The extended relational algebra, as presented in textbooks, consists of the well-known operators π (projection), σ (selection) and $\times$ (join) completed with the γ (grouping) together with the set theoretic operators. We focus on the former four operators. In our formalisation, `formula` mimics the SQL's filtering conditions expressed in the `where` and `having` clauses of SQL.

```
Inductive query : Type :=
 | Q_Table : relname → query
 | Q_Set : set_op → query → query → query
 | Q_Join : query → query → query
 | Q_Pi : list select → query → query
 | Q_Sigma : formula → query → query
 | Q_Gamma :
    list term → formula → list select →
      query → query
with formula : Type :=
 | Q_Conj :
    and_or → formula → formula → formula
 | Q_Not : formula → formula
 | Q_Atom : atom → formula
with atom : Type :=
 | Q_True
 | Q_Pred : predicate → list term → atom
 | Q_Quant :
    quantifier → predicate → list term →
      query → atom
 | Q_In : list select → query → atom
 | Q_Exists : query → atom.
```

We assume that there is an `instance` which associates to each relation a multiset (`bagT`) of tuples, and that these multisets enjoy some list-like operators such as `empty`, `map`, `filter`, etc (see the additional material for more details and precise definitions). In order to support so-called SQL correlated queries, the notion of environment is necessary.

```
Fixpoint eval_query env q {struct q} : bagT :=
  match q with
    | Q_Table r ⇒ instance r
    | Q_Set o q1 q2 ⇒ if sort q1 = sort q2
                      then interp_set_op o (eval_query env q1) (eval_query env q2)
                      else empty
    | Q_Join q1 q2 ⇒ natural_join (eval_query env q1) (eval_query env q2)
    | Q_Pi s q ⇒  map (fun t ⇒ projection_ (env_t env t) s) (eval_query env q)
    | Q_Sigma f q ⇒ filter (fun t ⇒ eval_formula (env_t env t) f) (eval_query env q)
    | Q_Gamma lf f s q ⇒ let g := Group_By lf in
       mk_bag (map (fun l ⇒ projection_ (env_g env g l) s)
               (filter (fun l ⇒ eval_formula (env_g env g l) f)
               (make_groups_ env (eval_query env q) g)))
  end
with eval_formula env f := [ ... ]
with eval_atom env atm := [ ...]
end.
```

Let us detail the evaluation of `Q_Sigma f q` in environment `env`. It consists of the tuples `t` in the evaluation of `q` in `env` which satisfy the evaluation of formula `f` in `env`. In order to evaluate `f` one has to evaluate the expressions it contains. Such expressions are formed with attributes which are either bound in `env` or occur in tuple `t`'s support. This is why the evaluation of `f` takes place in environment `env_t env t` which corresponds to pushing `t` over `env` yielding

```
Q_Sigma f q ⇒ filter (fun t ⇒ eval_formula (env_t env t) f) (eval_query env q)
```

Similarly, we use `env_t env t` for the evaluation of expressions of `s` in the `Q_Pi s q` case. The grouping γ is expressed thanks to `Q_Gamma`. A group consists of elements which evaluate to the same values for a list of grouping expressions. Each group yields a tuple thanks to the `list select` part in which each (sub-)term either takes the same value for each tuple in the group, or consists in an aggregate expression. This usual definition (see for instance [18]) is not enough to handle SQL's `having` conditions, as `having` directly operates on the group that carry more information than the corresponding tuple. This is why `Q_Gamma` has also a `formula` operand. Thus the corresponding expression for query

```
          select avg(a1) as avg_a1, sum(b1) as sum_b1 from t1
          group by a1+b1, 3*b1 having a1 + b1 > 3 + avg(c1);
```

is `Q_Gamma [a1 + b1; 3*b1] (Q_Atom (Q_Pred > [a1 + b1; 3 + avg(c1)]))`

```
      [Select_As avg(a1) avg_a1; Select_As sum(b1) sum_b1] (Q_table t1)
```

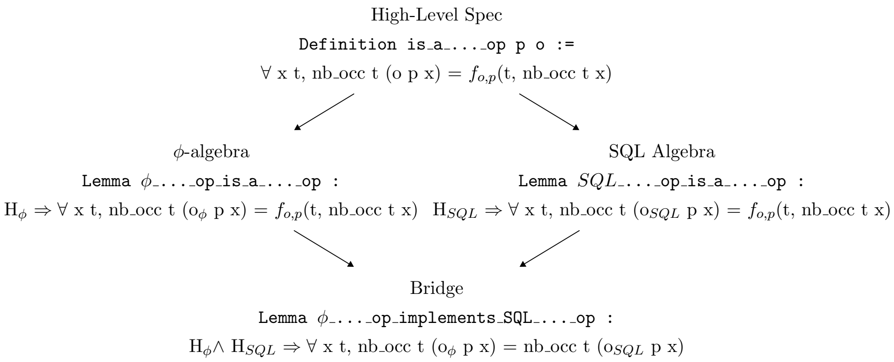
5.2 Adequacy

The following lemmas assess that SQL algebra is a realisation of our high-level specification. Note that, in the context of SQL algebra the notion of `tuple` corresponds to the high-level notion of elements' type `X`, finite bag corresponds to the high-level notion of `containerX` and `elements` to `contentX`.

```
Lemma Q_Sigma_is_a_filter_op : ∀env f,
    is_a_filter_op [...]
      (* contentA := fun q ⇒ Febag.elements BTupleT (eval_query env q) *)
      (* contentA' := fun q ⇒ Febag.elements BTupleT (eval_query env q) *)
      (fun t ⇒ eval_formula (env_t env t) f)
      (fun q ⇒ Q_Sigma f q).
Lemma Q_Join_is_a_join_op : ∀env s1 s2,
    let Q_Join q1 q2 := Q_Join q1 q2 in
    is_a_join_op (* contentA1 := fun q ⇒ elements (eval_query env q) *)
                 (* contentA2 := fun q ⇒ elements (eval_query env q) *)
                 (* contentA := fun q ⇒ elements (eval_query env q) *) [...] s1 s2 Q_Join.
Lemma Q_Gamma_is_a_grouping_op : ∀env g f s ,
    let eval_s l := projection_ (env_g env (Group_By g) l) (Select_List s) in
    let eval_f l := eval_formula (env_g env (Group_By g) l) f in
    let mk_grp g q := partition_list_expr (elements (eval_query env q))
                        (map (fun f t ⇒ interp_funterm (env_t env t) f) g) in
    let Q_Gamma g f s q := eval_query env (Q_Gamma g f s q) in
    is_a_grouping_op [...] mk_grp g eval_f eval_s (Q_Gamma g f s).
```

6 Formally bridging logical and physical algebra

We now formally bridge physical algebra to SQL algebra. Fig. 4 describes the general picture. As pointed out in Section 3, any two operators which satisfy the same high-level specification are interchangeable. This means in particular that physical algebra's operators can be used to implement the evaluation of constructors of SQL algebra's inductive `query`. The fundamental nature of the proof of such facts is the transitivity of equality of number of occurences. However, there are some additionnal hypotheses in both lemmas ϕ`_..._op_is_a_..._op` and `SQL_..._op_is_a_..._op`. Some of them are trivially fulfilled when the elements are tuples, while others cannot be discarded.



High-Level Spec

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

∀ 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_ϕ ⇒ ∀ 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} ⇒ ∀ x t, nb_occ t (o_{SQL} p x) = $f_{o,p}$(t, nb_occ t x)

Bridge

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

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

Fig. 4: Relating ϕ-algebra and SQL algebra.

For instance, for proving that `NestedLoop` implements `Q_Join`, we have to check that the hypotheses for `NL_is_a_filter_op` are fulfilled. Doing so, the condition that the queries to be joined must have disjoint sorts appeared mandatory in order to prove the hypothesis `build_split_eq_2` which assess that whenever two combined tuples are equivalent, their projections over the part corresponding to the inner relation also have to be equivalent.

```
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) = nb_occ t (Cursor.materialize C1 c1)) →
    (∀t, nb_occ t (eval_query env q2) = nb_occ t (Cursor.materialize C2 c2)) →
      (* ... then the nested loop implements the join *)
      ∀t, nb_occ t (eval_query env (Q_Join q1 q2)) =
        nb_occ t (Cursor.materialize NL (NestedLoop.mk_cursor C1 C2 nil c1 c2)).
```

This is an a posteriori justification that most systems implement combination of relations as cross-products whereas according to theory [1], combination should be the natural join.

7 Related works, lessons, conclusions and perspectives

Related work Our work is rooted on the many efforts to use proof assistants to mechanise commercial languages' semantics and formally verify their compilation as done with the seminal work on Compcert [23]. The first attempt to formalise the relational data model using Agda is described in [20,19] and a first complete Coq formalisation of it is found in [8]. A SSreflect-based mechanisation of the Datalog language has been proposed in [9]. The very first Coq formalisation of RDBMSs' is detailed in [24] where the authors proposed a verified source to source compiler for (a small subset) of SQL. In [14], an approach which automatically compiles high-level SQL-like specifications down into performant, imperative, low-level code is presented. Our goal is different as we aim at verifying real-life RDBMS's execution strategies rather than producing imperative code. More recently, in [3,4] a Coq modelisation of the nested relational algebra is provided to assign a semantics to data-centric languages among which SQL. Regarding logical optimisation, the most in depth proposal is addressed in [12] where the authors describe a tool to decide whether two SQL queries are equivalent. However, none of these works consider specifying and verifying the low-level aspects of SQL's compilation and execution as we did. Our work is, thus, complementary to theirs and one perspective could be to join our efforts along the line of formalising data-centric systems.

Lessons, conclusions and perspectives While formalising the *low level layer* of RDBMSs and SQL's *physical optimisation*, we learnt the following lessons: (i) not only finding the right invariants for physical operators was really involved but proving them (in particular termination for nested loop) was indeed subtle. This is due to the inherent difficulty to design on-line versions of even trivial off-line algorithms. (ii) we are even more convinced by the relevance of designing such a high-level specification that opens the way for accounting other data-centric languages. More precisely, we first formalised SQL algebra then the physical one, this implied revising the specification: in particular the introduction of `containersX` was made. Then, while bridging both formalisms we slightly modified the specification but without questionning our fundamental choices about abstracting over collections using `containersX`, only hypotheses were slightly tuned. (iii) The need for higher-order and polymorphism was mandatory both for the specification and physical algebra modelisation. This prevented us from using deductive verification tools such as Why3 [16] for instance: it was quite difficult to write down the algorithms and their invariants in this setting, even worse the automated provers were of no use to discharge the proof obligations. We tried tuning the invariants to help provers, without success. Hence our claim is that it is easier to directly use a proof assistant, where one has the control over the statements which have to be proven. (iv) The last point is that we experimented records versus modules: records are simpler to use than modules in our formalisation (no need of definitions' unfolding, no need of intermediate inductive types for technical reasons), the counterpart being that modules in the standard Coq library, such as `FSets` or `FMaps` were not directy usable. The nice feature which allows to hide part of their contents through module subtyping was not needed here.

There are many points still to be addressed. In the very short term we plan to specify the missing operators of Table 1 and enrich the physical algebra with more fancy algorithms. Along this line two directions remain to be explored. In our development, the emphasis was put on specification rather than performance. Even if we carefully separated functions used in specification (such as `collection`, `coherent`, . . .) from the concrete algorithms, these latter are defined in the functional language of Coq using higher-order data structures. We plan to refine these algorithms into more efficient versions, in particular that manipulate the memory. We plan to rely on CertiCoq [2] in order to produce fully certified C code. We are confident that our specification is modular enough to be plugged on other system components, such as buffer management, page allocation, disk access, already formalised in Coq as in [21,11]. Back to the general picture of designing a fully certified SQL compilation chain, in [6] we provided a Coq mechanised semantics pass that assigns any SQL query its executable SQL algebra expression. What remains to be done is to formally prove equivalence between SQL algebra expressions: those produced by the logical optimisation phase and the one corresponding to the query execution plan. Last, we are confident that our specification is general enough to host various data-centric languages and will provide a framework for data-centric languages interoperability which is our long term goal.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger-Savary, O., Sozeau, M., Weaver, M.: Certicoq: A verified compiler for coq. In: The Third International Workshop on Coq for Programming Languages (CoqPL) (2017)
3. Auerbach, J.S., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J.: Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suciu, D. (eds.) Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. pp. 1555–1569. ACM (2017). https://doi.org/10.1145/3035918.3035961, `http://doi.acm.org/10.1145/3035918.3035961`
4. Auerbach, J.S., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J.: Q*cert: A platform for implementing and verifying query compilers. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. pp. 1703–1706 (2017)
5. Bailis, P., Hellerstein, J.M., Stonebraker, M. (eds.): Readings in Database Systems, 5th Edition. MIT-Press (2015), `http://www.redbook.io/`
6. Benzaken, V., Contejean, E.: SQLCert: Coq mechanisation of SQL's compilation: Formally reconciling SQL and (relational) algebra (Oct 2016), working paper available on demand

7. Benzaken, V., Contejean, E.: A Coq mechanised executable algebraic semantics for real life SQL queries (2018), submitted for publication
8. Benzaken, V., Contejean, E., Dumbrava, S.: A Coq Formalization of the Relational Data Model. In: 23rd European Symposium on Programming (ESOP) (2014)
9. Benzaken, V., Contejean, E., Dumbrava, S.: Certifying standard and stratified datalog inference engines in ssreflect. In: Ayala-Rincon, M., Munoz, C. (eds.) 8th International Conference on Interactive Theorem Proving. vol. 10499. Springer (2017)
10. Chamberlin, D.D., Boyce, R.F.: SEQUEL: A structured english query language. In: Rustin, R. (ed.) Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes. pp. 249–264. ACM (1974). https://doi.org/10.1145/800296.811515, http://doi.acm.org/10.1145/800296.811515
11. Chen, H., Wu, X.N., Shao, Z., Lockerman, J., Gu, R.: Toward compositional verification of interruptible OS kernels and device drivers. In: Krintz, C., Berger, E. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 431–447. ACM (2016). https://doi.org/10.1145/2908080.2908101, http://doi.acm.org/10.1145/2908080.2908101
12. Chu, S., Weitz, K., Cheung, A., Suciu, D.: Hottsql: Proving query rewrites with univalent sql semantics. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 510–524. PLDI 2017, ACM, New York, NY, USA (2017)
13. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970). https://doi.org/10.1145/362384.362685, http://doi.acm.org/10.1145/362384.362685
14. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015. pp. 689–700 (2015)
15. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 2nd Edition. Benjamin/Cummings (1994)
16. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6, https://doi.org/10.1007/978-3-642-37036-6_8
17. Filliâtre, J.C., Pereira, M.: Itérer avec confiance. In: Journées Francophones des Langages Applicatifs. Saint-Malo, France (Jan 2016), https://hal.inria.fr/hal-01240891
18. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book (2. ed.). Pearson Education (2009)
19. Gonzalia, C.: Towards a formalisation of relational database theory in constructive type theory. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS. LNCS, vol. 3051, pp. 137–148. Springer (2003)
20. Gonzalia, C.: Relations in Dependent Type Theory. Ph.D. thesis, Chalmers Göteborg University (2006)

21. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: Certikos: An extensible architecture for building certified concurrent OS kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
22. Karp, R.M.: On-line algorithms versus off-line algorithms: How much is it worth to know the future? In: van Leeuwen, J. (ed.) Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992. IFIP Transactions, vol. A-12, pp. 416–429. North-Holland (1992)
23. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009)
24. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: ACM Int. Conf. POPL (2010)
25. Ramakrishnan, R., Gehrke, J.: Database management systems (3. ed.). McGraw-Hill (2003)
26. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1. pp. 23–34 (1979)
27. The Coq Development Team: The Coq Proof Assistant Reference Manual (2010), http://coq.inria.fr, http://coq.inria.fr
28. The Isabelle Development Team: The Isabelle Interactive Theorem Prover (2010), https://isabelle.in.tum.de/, https://isabelle.in.tum.de/