A Coq formalisation of SQL's execution engines

Véronique Benzaken, Évelyne Contejean, Chantal Keller, Eunice Martins

▶ To cite this version:

Véronique Benzaken, Évelyne Contejean, Chantal Keller, Eunice Martins. A Coq formalisation of SQL's execution engines. ITP 2018 - International Conference on Interactive Theorem Proving, Jul 2018, Oxford, United Kingdom. pp.88-107, 10.1007/978-3-319-94821-8_6 . hal-01716048

HAL Id: hal-01716048
https://hal.science/hal-01716048

Submitted on 23 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Coq formalisation of SQL's execution engines

V. Benzaken É. Contejean Ch. Keller
E. Martins
CNRS, 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 [25] or Isabelle [26] 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 [24] and detailed in [15].

The theoretical foundations for RDBMS's go back to the 70's where relational algebra was originally defined by Codd [11]. Few years later, SQL, the standard domain specific language for manipulating relational data was designed [9]. SQL was dedicated to *efficiently* retrieve data stored on *secondary storage* in RDBMS's, as described in the seminal work [24] 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 [15, 23, 12, 5] for instance). In particular, the semantic analysis and logical optimisation of a SQL query could yield an expression of an (extended) relational algebra. Then an evaluation strategy for the optimised algebraic expression, 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 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 goals, we first design a high-level, Coq specification for data-centric operators intended to capture their essence. Rather than choosing an extended algebra such as the ones presented in [15, 23, 22], we seeked for a very abstract, generic, thus extensible, specification as we plan to address other data models and languages than relational ones.

We, then, provide two implementations of our specification in Coq. The first one consists in low level operators as found in systems such as Postgresql and described in main textbooks on the topic [15, 23]. 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 ([19]) 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 implementation 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 [6]. We relate each algebraic operator to our high level specification by proving adequation 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 implements SQL algebra, we give strong assurances that the QEP and the algebraic expression resulting from the semantics analysis do have the same semantics. This last step has been eased thanks to the efforts devoted to the design of our high-level specification. 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.

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 [15] 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 [1], 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)[2] 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;
```

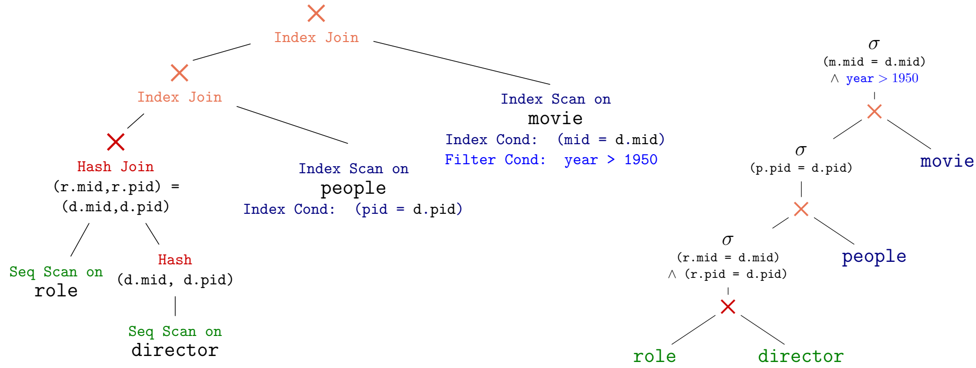


Figure 1: A typical SQL query, its QEP and logical AST.

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

[2]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 $x \times_{\texttt{IJ}} (\texttt{Index Scan}\ y\ \texttt{Index Cond :} a = x.a')$ 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.

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

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

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[3].

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 ∪ sa2)) ⋀
 (nb_occA t (j s1 s2) = nb_occA1 (projA1 t) s1 ∗ nb_occA2 (projA2 t) s2))
```

[3] 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.

∗ (if supA t = (sa1 ∪ 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 [15], 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.

```
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 $f_{o,p}$(t, nb_occX (g t) s), where $f_{o,p}$ 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 [15] and in the same spirit of the formalisation of cursors presented in [14], 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[4] named Cursor that contains (at least) an abstract type of cursors and these three operations:

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

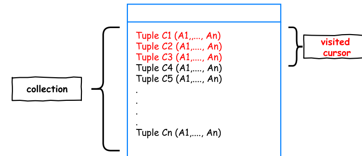
Due to the immutable nature of Coq objects, the operations next and reset must return the modified cursor. Moreover, since next must be a total function, a monadic construction is used to wrap the element of type t that it outputs:

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

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_collection : ∀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;
```

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

```
  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 of the cursor *)
  reset_visited : ∀c, visited (reset c) = nil;
  (* reset returns a coherent cursor *)
  reset_coherent : ∀c, coherent (reset c); [...]}.
```

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.

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 is stated as:

```
Definition coherent (c : cursor) : Prop :=
  Cursor.collection c = (rev (Cursor.visited c)) ++(to_visit c).
```

meaning 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 ubound (c:cursor) : nat := List.length (to_visit c).
```

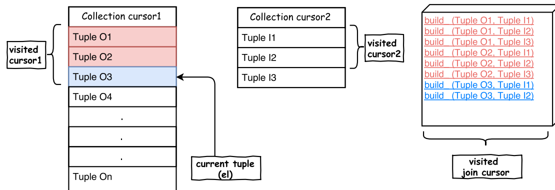
Second instance: filter Filtering a cursor returns the same cursor, but with a different function next and accordingly different specification functions. Given a property on the elements f : elt → bool, the function next filters elements of the underlying cursor:

```
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).



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

Figure 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 : cursor) :=
   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 nth 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).

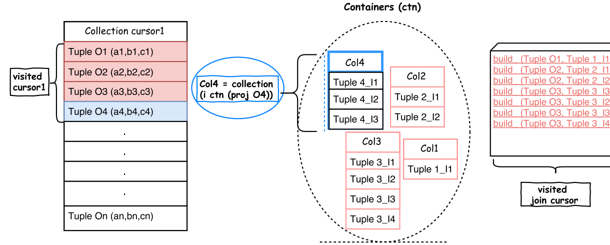
Derived operators Our high level of abstraction gives for free the specification of common variants of the physical operators. For instance, the Block Nested Loop algorithm is straightforwardly formalised by replacing, in the Nested Loop formalisation, the abstract type of elements by a type of "blocks" 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



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
```

Figure 3: Index-based nested loop

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 [15]) 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.

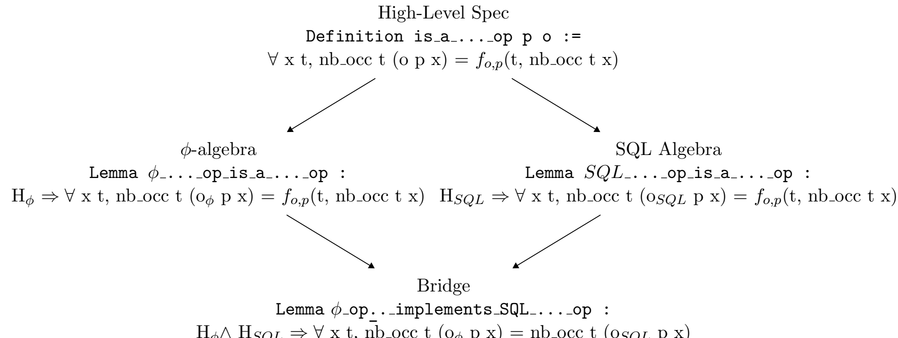


Figure 4: Relating ϕ-algebra and SQL algebra.

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 ∩ sort q2) = ∅ →
    (∀ 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 [20]. The first attempt to formalise the relational data model using Agda is described in [17, 16] and a first complete Coq formalisation of it is found in [7]. A SSreflect-based mechanisation of the Datalog language has been proposed in [8]. The very first Coq formalisation of RDBMSs' is detailed in [21] where the authors proposed a verified source

to source compiler for (a small subset) of SQL. 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 [10] 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 We used the Coq proof assistant to *specify* and *verify* the *low level layer of an RDBMS* as well as SQL's compilation's *physical optimisation*: fundamental steps towards mechanising SQL's compilation chain.

While formalising all this, 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 [13] 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. In [18] an Isabelle formalization of the complexity of on-line algorithms is proposed and we shall rely on it to formally assess the complexity of the algorithms presented so far. 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. 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] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. Bélanger-Savary, M. Sozeau, and M. Weaver. Certicoq: A verified compiler for coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[3] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1555–1569. ACM, 2017.

[4] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. 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*, pages 1703–1706, 2017.

[5] P. Bailis, J. M. Hellerstein, and M. Stonebraker, editors. *Readings in Database Systems, 5th Edition.* 2015.

[6] V. Benzaken and E. Contejean. A Coq mechanised executable algebraic semantics for real life SQL queries. Submitted for publication, 2018.

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

[8] V. Benzaken, E. Contejean, and S. Dumbrava. Certifying standard and stratified datalog inference engines in ssreflect. In M. Ayala-Rincon and C. Munoz, editors, *8th International Conference on Interactive Theorem Proving*, volume 10499. Springer, 2017.

[9] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured english query language. In R. Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, pages 249–264. ACM, 1974.

[10] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent sql semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 510–524, New York, NY, USA, 2017. ACM.

[11] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[12] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.

[13] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *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*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[14] J.-C. Filliâtre and M. Pereira. Itérer avec confiance. In *Journées Francophones des Langages Applicatifs*, Saint-Malo, France, Jan. 2016.

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

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

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

[18] M. P. L. Haslbeck and T. Nipkow. Verified Analysis of List Update Algorithms. In A. Lal, S. Akshay, S. Saurabh, and S. Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[19] R. M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In J. van Leeuwen, editor, *Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7-11 September 1992*, volume A-12 of *IFIP Transactions*, pages 416–429. North-Holland, 1992.

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

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

[22] G. Moerkotte. Building query compilers. http://pi3.informatik.uni-mannheim.de/ moer/querycompiler.pdf.

[23] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[24] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 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.*, pages 23–34, 1979.

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

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

A Running example's complete SQL definitions

A.1 Database schema and instance

The complete database schema for the introductory example is given Figure 5. It models a movie database gathering information about persons (`people`, the name of the relation) who have a `firstname` and `lastname` and are uniquely identified by a personal identifier `pid`. In the relational database setting, `firstname`, `lastname` and `pid` are called *attributes* and constitute the *sort* of the *relation schema*: `people(pid, firstname, lastname)`. Each relation contains (a finite number of) *tuples* which are records labelled by the sort of the relation they belong to.

```
create table people
(pid integer,
firstname VARCHAR(30),
lastname VARCHAR(30),
PRIMARY KEY(pid));
```

```
create table movie
(mid integer,
title VARCHAR(90) not Null,
year integer not Null,
runtime integer not Null,
rank integer not Null,
PRIMARY KEY(mid));
```

```
create table director
(mid integer REFERENCES movie,
pid integer REFERENCES people,
PRIMARY KEY (mid, pid),
FOREIGN KEY (mid)
       REFERENCES movie,
FOREIGN KEY (pid)
       REFERENCES people);
```

```
create table role
(mid integer,
pid integer,
name VARCHAR(70),
PRIMARY KEY(mid, pid, name),
FOREIGN KEY (mid)
       REFERENCES movie,
FOREIGN KEY (pid)
       REFERENCES people);
```

Figure 5: A movie database

Declaration `PRIMARY KEY(pid)` states that no two persons can have the same `pid` and different first and last names. Movies (`movies`) have a `title`, a `year` of release, a duration (`runtime`) and a `ranking`. As for relation `people`, movies are uniquely identified thanks to a movie id (`mid`). Relation `director` gathers information about movies' directors and the movies they directed. Declaration `FOREIGN KEY (mid) REFERENCES movie` indicates that any `mid` must also be present in `movie`. Last, relation `role` carries information about who played (identified by his/her `pid`) which role (represented by `name`) in a given movie (identified by its `mid`).

While those declarations describe relationships between entities at a logical level, they do have an impact at a system level. The `create table` declaration induces the creation of a file intended to store the elements of its argument. More importantly, the primary key definition induces the creation of a tree-structured index [5] whose search key is `pid` (resp., `mid`, (`mid, pid`), (`pid, mid, name`)). Such an auxiliary data structure could be exploited in order to speed up access to data stored in the corresponding indexed files.

[5] In this particular case it is a B-tree, a very popular data structure used in most, if not all, RDBMS's.

A.2 Example's query and its Postgresql QEP explained

Our SQL query extracts the directors who have played in a movie they directed after 1950. Relational algebra expression corresponding to our query is:

```
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;
```

Figure 6: A typical SQL query

$$\sigma_{\texttt{m.mid=d.mid} \wedge \texttt{year} > 1950 \wedge \texttt{p.pid=d.pid} \wedge \texttt{r.mid=d.mid} \wedge \texttt{r.pid=d.pid}}(\texttt{people} \times \texttt{director} \times \texttt{role} \times \texttt{movie})$$

where σ and $\times$ represent the *multiset* version of the well-known *selection* and *cross product* (or *join*) of the relational algebra as found in textbooks. The compiler exploits well-known laws such as associativity, and commutativity of operators and logical connectives, pushing selection and many others to produce:

$$\sigma_{\texttt{m.mid=d.mid}}(\sigma_{\texttt{p.pid=d.pid}}(\sigma_{\texttt{r.mid=d.mid} \wedge \texttt{r.pid=d.pid}}(\texttt{role} \times \texttt{director}) \times \texttt{people}) \times \sigma_{\texttt{year}>1950}(\texttt{movie}))$$

As previously stated, once the logical query plan is obtained a physical query evaluation plan is produced. To do so, the optimiser relies on statistics (histograms, distribution laws) about data, such statistics being collected along the time. Then roughly, each plan is assigned a cost thanks to very sophisticated methods (simulated annealing, dynamic programming, hill climbing) and the cheapest plan is chosen. Any decent system, such as Postgresql through the `explain` command, or Oracle™ through the `explain plan` command, allows us to see which is the physical query plan chosen for a given query.

```
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;

                                         QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=32.00..448.85 rows=2 width=7)
   ->  Nested Loop  (cost=32.00..432.28 rows=2 width=15)
         ->  Hash Join  (cost=32.00..415.72 rows=2 width=16)
               Hash Cond: ((r.mid = d.mid) AND (r.pid = d.pid))
               ->  Seq Scan on role r  (cost=0.00..238.35 rows=14535 width=8)
               ->  Hash  (cost=15.80..15.80 rows=1080 width=8)
                     ->  Seq Scan on director d  (cost=0.00..15.80 rows=1080 width=8)
         ->  Index Scan using people_pkey on people p  (cost=0.00..8.27 rows=1 width=11)
               Index Cond: (pid = d.pid)
   ->  Index Scan using movie_pkey on movie m  (cost=0.00..8.27 rows=1 width=4)
         Index Cond: (mid = d.mid)
         Filter: (year > 1950)
(12 rows)
```

Figure 7: Example of a (Postgresql) physical query plan

Figure 7 shows the Postgresql plan proposed for our query. This plan reflects the evaluation strategy chosen by the compiler and depends on the data actually stored. 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 and last the estimated costs of execution. Traditional practice is to measure the costs in units of disk page fetches. It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It is also important to realise that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan.

The rows value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the number scanned, as a result of filtering by any `where` clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

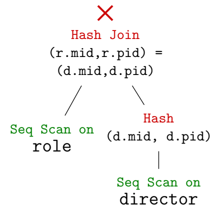
Our QEP on Figure 7 deserves more explanations. Even if each system has its own notation, query physical plans are built from *pre-defined access methods* to relations and *pre-defined (physical) operators* each of which implements one step of the plan. All the process starts with secondary storage access and the compiler is in charge of choosing how data stored in relations `role, director, movie` and `people` are to be accessed.

A.2.1 The first join

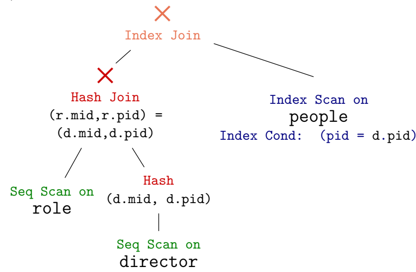
First, consider the left-most leaf of the physical plan of Figure 7 represented on Figure 8a. Relations `director` and `role` will be accessed through a *sequential scan* of the respective file (`Seq Scan on director` and `Seq Scan on role`). `Seq Scan` is one, among many others, *access methods* and consists in reading each record present in the file that stores the relation it has as parameter. While scanning relation `director` the system generates a hash table (`Hash`) on attributes (`d.mid, d.pid`) so as to evaluate the cross product using a hash-join algorithm (`Hash Join Hash Cond: ((r.mid = d.mid) AND (r.pid = d.pid))` which corresponds to relational expression:

$$\sigma_{\texttt{r.mid}=\texttt{d.mid}\wedge\texttt{r.pid}=\texttt{d.pid}}(\texttt{director}\times\texttt{role})$$

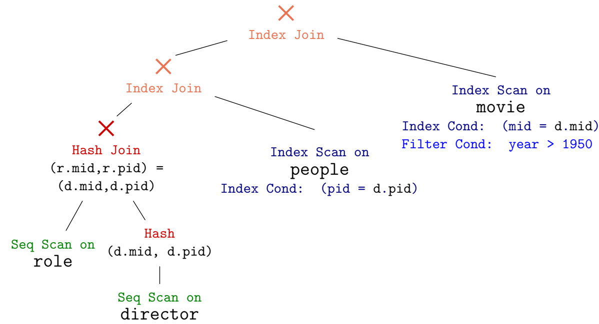
`Hash Join` is one of the *physical operators* usually provided by RDBMS's to implement a join and falls in the general class of *index-based* operators. Not only the optimiser chooses the access methods and algorithm to implement the join but it also relies on the fact that operator $\times$ is associative and commutative thus allowing for the evaluation of `director` $\times$ `role` first.



(a) Intermediate physical query plan for the first join



(b) Intermediate physical query plan for the second join



(c) Final physical query plan

Figure 8: Query plan explained

A.2.2 The second join

Then, tuples resulting of the evaluation of the `Hash Join` are planned to be passed *on the fly* to the upper operator, which is *dependent on* the evaluation of the first join. In particular each `d.pid` will serve as a constant and the index on relation `people` whose search key is `pid` will be exploited to access `people` elements by means of an *index scan* (`Index Scan using people_pkey on people p`) with the (index) search condition `Index Cond: (pid = d.pid)`. Indeed, the compiler does exploit the knowledge that an index, named `people_pkey`, on relation `people` whose search key is `pid` is available. The cross product between both collections is achieved using the nested loop physical operator join algorithm (`Nested Loop` on the postgresql plan) combined with the `Index Scan` on people. The combination of both physical operators implements an *index based* join (`IndexJoin`) (see Figure 8b) which corresponds to relational expression:

$$\sigma_{\texttt{p.pid}=\texttt{d.pid}}(\sigma_{\substack{\texttt{r.mid} = \texttt{d.mid} \\ \wedge \\ \texttt{r.pid} = \texttt{d.pid}}} (\texttt{director} \times \texttt{role}) \times \texttt{people})$$

A.2.3 The third join

Last `movie`'s elements are to be accessed exploiting the index (`Index Scan using movie_pkey on movie m Index Cond: (mid = d.mid)`) and *at the same time* elements that do not satisfy the filtering condition `Filter: (year > 1950)` will be discarded. The combination of the corresponding elements is performed again through a nested loop algorithm (`Nested Loop`) yielding physical and logical plans of Figure 8c. We would like to stress that, the computation, again, depends on the previously computed value d.mid. A last subtle point worth to be mentionned is that rather than implementing:

$$\sigma_{\texttt{m.mid}=\texttt{d.mid}}(\sigma_{\texttt{p.pid}=\texttt{d.pid}}(\sigma_{\texttt{r.mid}=\texttt{d.mid} \wedge \texttt{r.pid}=\texttt{d.pid}}(\texttt{role} \times \texttt{director}) \times \texttt{people}) \times \sigma_{\texttt{year}>1950}(\texttt{movie}))$$

the plan implements rather

$$\sigma_{\texttt{m.mid}=\texttt{d.mid} \wedge \texttt{year}>1950}(\sigma_{\texttt{p.pid}=\texttt{d.pid}}(\sigma_{\texttt{r.mid}=\texttt{d.mid} \wedge \texttt{r.pid}=\texttt{d.pid}}(\texttt{role} \times \texttt{director}) \times \texttt{people}) \times \texttt{movie})$$

B Adequacy and bridge lemmas for `Filter` operators

Let us now explain how we use our high level specification in order to show that the algebraic operator Q_Sigma is implemented by some physical operators, namely the simple Filter and IndexScan when the selection condition can be expressed thank to an equality between index keys, keys being usually a list of values taken by some attributes of the tuples.

First, assuming that we have a cursor c which contains elements of type A, we can filter its content w.r.t theta by using Filter, provided that theta is compatible w.r.t. the equivalence defined by the decidable comparison function in OA.

```
Section Filter_is_a_filter_op.
  Hypothesis A : Type.
  Hypothesis OA : Oeset.Rcd A.
  Hypothesis C : Cursor.Rcd OA.
  Hypothesis theta : A → bool.
  Hypothesis theta_eq : ∀x1 x2, Oeset.compare OA x1 x2 = Eq → theta x1 = theta x2.

  Lemma mk_filter_is_a_filter_op :
    let F := (Filter.build _theta_eq C) in
    is_a_filter_op
    (* a record containing a deciable comparison function over the elements to be selected *)
       OA
    (* how to retrieve the elements to be selected from the input container *)
       (Cursor.materialize C)
    (* how to retrieve the selected elements from the output container *)
       (Cursor.materialize F)
    (* selection condition *)
       theta
    (* the actual filter, which produced the output container from the input container *)
       (Filter.mk_filter F).
End Filter_is_a_filter_op.

Section IndexScan_is_a_filter_op.
  Variable o1 o2 : Type.
  Variable O1 : Oeset.Rcd o1.
  Variable O2 : Oeset.Rcd o2.
  Variable IS : Index.Rcd O1 O2.

Lemma IS_is_a_filter_op :
    ∀(x1 : o1),
    is_a_filter_op
    (* a record containing a deciable comparison function over the elements to be selected *)
       O1
    (* how to retrieve the elements to be selected from the input container *)
       (Index.c1 IS)
    (* how to retrieve the selected elements from the output container *)
      (Cursor.materialize (Index.C1 IS))
    (* selection condition: having the same key as [o1] *)
       (fun x ⇒ Index.P IS (Index.proj IS x1) (Index.proj IS x))
    (* the actual filter, which produced the output container from the input container *)
       (fun c ⇒ Index.i IS c (Index.proj IS x1)).
End IndexScan_is_a_filter_op.
```

Assuming that all needed ingredients (tuples with a decidable comparison, functions, predicates, aggregates with their respectives interpretations, etc.) are present to define the evaluation of an algebraic query, Q_Sigma is also is_a_filter_op:

```
Lemma Q_Sigma_is_a_filter_op :
  ∀env f,
    is_a_filter_op
    (* a record containing a deciable comparison function over the elements to be selected *)
```

```
    (OTuple T)
  (* how to retrieve the elements to be selected from the input container *)
    (fun q ⇒ Febag.elements BTupleT (eval_query env q))
  (* how to retrieve the selected elements from the output container *)
    (fun q ⇒ Febag.elements BTupleT (eval_query env q))
  (* selection condition: satisfying the formula [f] *)
    (fun t ⇒ eval_formula (env_t env t) f)
  (* the actual filter, which produced the output container from the input container *)
    (fun q ⇒ Q_Sigma f q).
```

It is now easy to bridge the above lemmas to show that **fun** q ⇒ Q_Sigma f q can always be implemented by a simple Filter built over a cursor which implements the algebraic query q and using as a filtering condition the evaluation of the formula f.

```
Notation "b '=R=' l" :=
  (∀ t, Febag.nb_occ BTupleT t b = Oeset.nb_occ (OTuple T) t l) (at level 70, no associativity).
Lemma mk_filter_implements_Q_Sigma :
  ∀ C env f q,
    let F := Filter.build _(eval_f_eq env f) C in
    ∀ c,
      eval_query env q =R= Cursor.materialize C c →
      eval_query env (Q_Sigma f q) =R= Cursor.materialize F (Filter.mk_filter F c).
```

When the formula f has the special form of a conjunction of equalities for the values taken by some attributes, as in our example , it is also possible to use an IndexScan:

```
Lemma IS_implements_Q_Sigma :
  ∀ (* the keys of the index are lists of values *)
    (IS : Index.Rcd (OTuple T) (oeset_of_oset (mk_olists (OVal T)))),
    (* the condition used to build the index is actually the equality of these lists of values *)
    (∀ lv1 lv2, Index.P IS lv1 lv2 = Oset.eq_bool (mk_olists (OVal T)) lv1 lv2) →
    ∀ la lval env f q ,
    (* the selection formula of the algebraic query corresponds to the equality between a key [lval]
       and the dot extraction over a given list of attributes [la] *)
    (∀ t, eval_formula (env_t env t) f =
                Oset.eq_bool (mk_olists (OVal T)) (map (dot T t) la) lval) →
    (* the projection used to build the index is actually the dot extraction over [la] *)
    (∀ t, Index.proj IS t = map (dot T t) la) →
    ∀ c,
      eval_query env q =R= Index.c1 IS c →
      eval_query env (Q_Sigma f q) =R= Cursor.materialize _(Index.i IS c lval).
```

The same kind of arguments hold for the other operators and are detailed in the accompanying development at `http://datacert.lri.fr/sqlcert/itp18.tar.gz`.

C Proof of concept: the example QEP

We have all the ingredients to design a language for query execution plans, that is interpreted using our cursors and indices from Section 4. We focus in this appendix on its subset applied to the example of Fig. 1.

C.1 A QEP language

The language is a syntactic tree where nodes are labeled with the physical operators, divided in two categories: simple cursors, and index-based iterators. Restricted to operators that are needed for the example of Fig. 1[6], the definition of the language contains the two inductive types:

```
(* o is the type of values *)

Inductive index : Type :=
| SimpleHashIndexScan : list attribute → list o → index
| FilterHashIndexScan :
    ∀(theta : o → bool), list attribute → list o → index.

Inductive cursor : Type :=
| SeqScan : list o → cursor
| IndexJoin : list attribute → cursor → index → cursor.
```

Indices can be mapped either directly to a sequential scan (as for the `Index Scan on people`), or to a filter over a sequential scan (as for the `Index Scan on movie`). In each case, the list of attributes represents the index. Cursors can be sequential scans or the combination of an outer cursor with an inner index.

This language is interpreted in two steps. First, all the physical operators are mapped to their `index` and `cursor` counterparts presented in Section 4. Second, this mapping is used to type (in Coq) the evaluation function that returns the concrete `index` or `cursor`.

```
Definition type_index (i:index) : Index.Rcd _ _:=
  match i with
  | SimpleHashIndexScan la _ ⇒ HashIndexScan.simple_build (proj la) (proj_eq _)
  | FilterHashIndexScan theta la _ ⇒
      HashIndexScan.filter_build _ (theta_eq theta) (proj la) (proj_eq _)
  end.
Fixpoint type_cursor (c : cursor) : Cursor.Rcd _:=
  match c with
  | SeqScan l ⇒ SeqScan.build _
  | IndexJoin la c i ⇒
      IndexJoin.build
      (proj la) (proj_eq la) (type_cursor c) (type_index i) build build_eq_1 build_eq_2
  end.

Definition eval_index (i:index) : Index.containers (type_index i) :=
  match i return Index.containers (type_index i) with
  | SimpleHashIndexScan la l ⇒
      HashIndexScan.mk_simple_hash_index_scan l _ (proj la) (proj_eq _)
  | FilterHashIndexScan theta la l ⇒
      HashIndexScan.mk_filter_hash_index_scan _ (theta_eq theta) l _ (proj la) (proj_eq _)
  end.
Fixpoint eval_cursor (c:cursor) : Cursor.cursor (type_cursor c) :=
```

[6] A `Hash` operator applied to a sequential scan can be modeled as a `Hash Index Scan` where indices are map to sequential scans.

```
match c return Cursor.cursor (type_cursor c) with
| SeqScan l ⇒ SeqScan.mk_seqcursor l
| IndexJoin la c i ⇒
    IndexJoin.mk_index_join_cursor
    (type_cursor c) (type_index i) (eval_cursor c) (eval_index i)
end.
```

C.2 The example QEP

The query execution plan represented in Fig. 1 can now be syntactically expressed in this language. We define the base relations to contain a few movies and people, as a list of tuples, and the plans is:

```
Definition qep :=
  IndexJoin (d_mid :: nil)
    (IndexJoin (d_pid :: nil) ... ...)
    (FilterHashIndexScan theta_movie (m_mid :: nil) movies).
```

The evaluation of this plan uses the on-line algorithms presented in Section 4[7].

```
Definition qep_run :=
  Eval compute in (map show_tuple (materialize _(eval_cursor qep))).
```

[7] The result of the evaluation can be checked by compiling the auxiliary materials.