Contributions to a Verified Database Management System

Internship report - MPRI research master program

Pablo Le Hénaff

Supervisors: Andrew Appel, Lennart Beringer

Princeton University, Princeton, NJ 08540, United States of America

April 1st - August 31st, 2019

General context

Software verification is the area of computer science research devoted to the study and development of safe programming technology. Safe programs are free of bugs: they have no unexpected behavior and comply with their specification. Formal proof systems like *the Coq proof assistant* are computer applications that have been developed to express statements in a precise, logical, and mathematical language, and check their correctness.

The *Verified Software Toolchain* (VST) is a research project that explores the end-to-end formal verification of programs written in C. VST includes *Verifiable C* [8], a tool to prove the correctness of C code using *separation logic*, embedded inside Coq and developed at Princeton University.

Relational database management systems (RDBMS) are programs that are used to store large amounts of data organised in tables. They are widespread in data-intensive computer applications, such as web servers. A RDBMS can handle a wide variety of requests from the client, for instance to retrieve filtered or sorted data, or combine multiple sources of information. Such requests are expressed in a query language, the most well-known of them being SQL (for *Structured Query Language*). Queries are compiled down in several stages to a combination of efficient low-level algorithms by the RDBMS.

The topic of this report lies at the intersection of RDBMS engineering and state-of-the-art software verification using Coq and Verifiable C.

Research problem

Despite their omnipresence, there currently exists no usable, efficient RDBMS that has been formally guaranteed to execute safely.

RDBMSs used nowadays in real-world applications are highly complex and tightly engineered systems, thus proving their functional correctness using current techniques is impossible [2].

DeepSpecDB is an ongoing effort to program some components of a SQL RDBMS in efficient C and explore their VST verification.

My contributions

I have made contributions to DeepSpecDB on the following aspects:

– I reviewed current research work on SQL and RDBMS Coq formalization for use as functional model (section 1.2);

– I implemented in C the *index join* algorithm and some other algorithms along the way (section 2);

– I achieved the VST verification of a hashtable program, mapping integer keys to pointer values, that was used in my C code; I instantiated for that purpose a variant of a recently published separation logic technique (section 3);

– I formalized in Gallina the invariants and operations of the *B+tree* data structure at the heart of our system, and proved their properties (section 4).

Arguments supporting their validity

All the algorithms and data structures at stake are used in commercial, industry-grade systems and have shown many proofs of their efficiency for real-world applications. The only known equivalent project, DataCert [2], has been developing in Coq the upper part of a verified query compilation pipeline. They plan on compiling that dependently-typed code down to assembly using CertiCoq [6], a technology that is not ready yet and whose main focus is not performance.

DeepSpecDB focuses on the last stage of the pipeline: low-level data management. Writing C code directly and using Verifiable C to give correctness guarantees is a better approach here, because performance is crucial. Using the low-level memory management features of C, it is possible to write byte-precise optimized data manipulating algorithms.

Summary and future work

My contributions have advanced the current state of DeepSpecDB, both of the implementation and verification sides.

On the short term, the next task will be to complete the modular VST verification of the various data structures and algorithms that are already implemented. More RDBMS algorithms should be added: my code only implements the basic ones, while commercial RDBMSs have a wide panel of algorithms to choose from to optimize a request.

Finally, the low-level algorithms could be interfaced with a program that parses SQL queries and generates query execution plans. That program could be a commercial RDBMS optimizer paired with a verified plan checker using work from DataCert on SQL semantics, to reach a completely verified system usable for real-world tasks.

Notes and acknowledgments

This work was funded by *DeepSpec*[1], an *Expedition in Computing* funded by the *National Science Foundation*[2]. I also received a stipend from *École polytechnique* as a student of the *Ingénieur polytechnicien* program.

I am thankful to Professor Andrew Appel for inviting me to Princeton University, for his enlightening supervision and for the appreciated financial support. Thank you Lennart Beringer for your kind help. Thank you Anastasiya Kravchuk-Kirilyuk for being such a cheerful collaborator. I really enjoyed getting to know the programming languages group and learnt a lot during my time in Princeton.

[1] `www.deepspec.org`
[2] `www.nsf.gov`

1 Background and state of the art

1.1 Verifiable C

Verifiable C is a Coq framework and package for proving the functional correctness of C code in an interactive way. It implements a program logic for C. That logic is a kind of separation logic which allows to reason about imperative programs with pointers one statement at a time. Full correctness of an executable binary compiled from C can be achieved using VST the following way: Verifiable C is used to check the correctness of the C code using Coq and all the Verifiable C automation library, while the *CompCert certified compiler* generates assembly code that is guaranteed to satisfy the specifications of the C language.

That pipeline is worthy of trust since Verifiable C's logic has been mecanically proved sound with respect to the C semantics standard implemented by CompCert. This proof of soundness can be interpreted as in the following statement:

Any behavior proven using Verifiable C on a C program
is guaranteed to be a behavior of the resulting assembly code generated by CompCert.

Correctness proofs using Verifiable C consist in matching a low-level, imperative C function with a high-level, functional *specification* taking the form of a Hoare triple $\{P\}s\{Q\}$, where P and Q are Verifiable C predicates. That triple means that when the heap, global and local variables of the program are in a state satisfying predicate P, if statement s is executed by the machine and terminates, the heap and variables will be in a state satisfying Q afterwards. That specification involves the *functional model*, an implementation in Gallina of C data structures and functions.

Functional languages like Coq are intrinsically easier to reason about than C, because of the immutable nature of their variables: the notion of heap does not exist. Moreover, the *calculus of inductive constructions* and *dependent types* implemented by Coq are used to express and prove properties of Coq functions.

Therefore, proving correctness of C code using Verifiable C amounts not only to proving that the C program is a refinement of a functional model expressed in Gallina: once the high-level specification is proved correct, it is possible to prove further correctness properties on that functional model using Coq, which by transitivity will also hold on the generated binary.

Any Coq definition or library can be used as a functional model. However, Verifiable C promotes some standard choices: for instance, it is strongly advised to use the `Z` type of integers, even for non-negative numbers (as opposed to Peano natural numbers encoded with the successor function), as it makes arithmetic reasoning easier and more efficient.

1.2 Relational DBMS

A database management system executes requests issued by the client to query, add, alter, remove, or reorganise data stored in a database.

DeepSpecDB falls in the category of *relational* DBMS: we will only consider the relational model of data, which is the most commonly used and studied. Newer DBMS can accomodate data in other forms, such as graph data or semi-structured data like XML documents. These approaches are nowadays referred to by the concept of *NoSQL*.

RDBMS design involves both a (i) theoretical, abstract description of the data - the relational data model - that serves to compile queries into an (ii) algorithmic, concrete implementation of data management operations. This section's explanations will be interleaved with short descriptions of recent research

```
Fixpoint Tuple (A: Schema): Set :=
match A with
| nil ⇒ unit
| n:: t ⇒ ⟦n⟧ ∗ Tuple t
end.
```

```
SELECT 0, 1, 2 FROM
  (JOIN tbl1 , tbl2 ON col 0 = col 0)
WHERE col 0 = "hello world"
      AND col 1 < col 3
```

Figure 1: A first tuple formal model and a sample SQL-like query from [18]

work in the formal methods community that I have studied to understand what formal description of some database system components we could use to specify our DeepSpecDB system.

In particular, *DataCert* is an ongoing research project at the intersection of formal methods, the theory of databases and their functional implementation. It is being developed in Lille, Lyon, and Orsay in France and is funded by the Agence Nationale de la Recherche. DataCert plans to interact with the DeepSpec Expedition in Computing by using the CertiCoq verified compiler to generate binary executables [3].

1.2.1 Relational model and algebra

In the relational model, data is organized into *relations*, that are *multisets* of *tuples* sharing the same *attributes*. A multiset, or *bag*, is a set that can contain multiple occurences of a same element. The number of occurences of a given element is called its *multiplicity*. In a relation, the ordered list of attributes common to all tuples is called the *schema* in the database jargon. Each attribute has an associated *domain* with corresponds to the Coq notion of `Type`. Intuitively, tuples are rows in a table, the attributes (or fields) are columns of the table and the schema is the header, or specification of the columns. Each element in a tuple must belong to the domain of the corresponding attribute in the schema.

Malecha et al. [18] use a Coq inductive definition for tuples, reproduced in figure 1, based on a list of Coq types called `Schema`. Also included is a query in their SQL-like language, illustrating how fields are selected using their position in the schema.

Benzaken et al. [12] define another version of a tuple type. It is reproduced in figure 2. A tuple record `Tuple.Rcd` holds Coq types for relational data types, attributes, values and abstract tuples. Thus each attribute is identified by a distinct name in the `attribute` type. The `support` of a tuple is the finite set of attributes it covers. The `dot` function is projection on a single attribute. The `mk_tuple` parameter abstractly builds a tuple from its support and dot/projection function.

While it looks more complex than the previous tuples, this formalization of relational tuples is far more realistic. Indeed, the main flaw of the previous formalization is that attributes are denoted by their position in the tuple's schema, not by a given identifier: this is called the *unnamed* relational model. DataCert implements a *named* version of the relational model. This version is also implemented in all commercial RDBMSs. While the named and unnamed relational models are theoretically equivalent [12], the unnamed version does not allow attributes to carry semantic information nor does it allow RDBMSs to choose optimized algorithms and indexing data structures based on a given attribute.

Relations are defined in both cases as bags of tuples respectful of the relation's schema.

The *relational algebra* is an abstract algebraic structure that describes selection of relational data. The most commonly used relational algebra operators and their properties are thoroughly described in Garcia-Molina et al. [17] and summed up in figure 3, where R and R' are relations. The grouping and renaming operators (γ and ρ respectively), as well as exotic variants of join, are omitted for concision. The join operator we consider is a restricted version of the *theta-join*, that is, it accepts an equality predicate to identify attributes with potentially different name. This type of join is called the *equijoin*, and the attributes that are on both sides of the equality in the predicate are called *foreign keys*.

```
Module Tuple.
Record Rcd : Type := mk_R {
  [...]
  A : Fset.Rcd attribute;
  tuple : Type;
  support : tuple → set A;
  dot : tuple → attribute → value;
  mk_tuple : set A → (attribute → value) → tuple;
  [...]    }
```

Figure 2: The more realistic DataCert tuples, from [12]

Operator	Meaning
$\sigma_p(R)$	Selects tuples in R that satisfy predicate p
$\pi_{A_1,\ldots,A_n}$	Projects tuples in R on attributes $A_1, \ldots, A_n$
$R \bowtie_{\bigwedge_{i=1}^{n} R.A_i = R'.B_i} R'$	Joins R and R' by identifying for $i \in \{1, \ldots, n\}$ field A_i from R with field B_i from R'
$\cup$, $\cap$, $\setminus$	Usual set operations with bag semantics

Figure 3: Relational algebra operators

After being parsed into an abstract syntax tree, a SQL request is interpreted by the RDBMS as a term of the relational algebra. DataCert is the most advanced project providing detailed and complete mechanized semantics for the relational algebra [12] as well as for SQL [11] [13].

To illustrate relational algebra operators and their meaning, figure 4 shows a typical example of two tables sharing some information (fields **N** in A and **n** in B) that are being joined. The result is filtered and projected to execute a SQL query. One can notice that the equijoin is a somewhat silly operator, as it produces a new relation where multiple columns (**n** and **k** in the example) contain the exact same information under different attribute names. This behavior is the one implemented in commercial RDBMSs.

The *natural join* will join those attributes from both relations that have exactly the same name, and merge the resulting identical columns. My implementation of the index join, described in section 2, is made smarter than the traditional equijoin by merging identical columns like the natural join. The attribute name of the new merged column is the one of the left relation. It is unclear whether this behavior is a good thing or not; anyway, switching to the silly version is only a matter of deleting a few lines of code.

1.2.2 Query execution

Given a relational algebra term, the DBMS elaborates a concrete *query execution plan* (QEP). A QEP takes the form of a binary tree whose nodes are *physical, low-level algorithms* that effectively treat and output the data in memory. A same relational algebra term is implemented by several QEPs, with different expected running times. These running times vary in particular with the size of tables: a small relation will be more efficiently treated as is, while a large relation may benefit from building dedicated indexing structures. The RDBMS tries to choose the fastest QEP to execute a query. A detailed exposition of PostgreSQL QEPs can be found in [21], section 14.1: *Using EXPLAIN*. The QEP generated by PostgreSQL for the query presented in figure 4 is detailed in figure 5.

Notice how the join comes last in the QEP while it came first in the relational algebra term. Indeed, multiple abstract terms (respectively QEPs) are available to describe (resp. execute) a same query.

a	
n	**lit**
0	zero
1	one
2	two
3	three

b	
k	**even**
0	T
1	F
2	T
3	F
4	T

a $\bowtie_{\text{a.n=b.k}}$ b			
n	**lit**	**k**	**even**
0	zero	0	T
1	one	1	F
2	two	2	T
3	three	3	F

$\sigma_{\text{even=T}}$(a $\bowtie_{\text{a.n=b.k}}$ b)			
n	**lit**	**k**	**even**
0	zero	0	T
2	two	2	T

$\pi_{\text{n,lit}}(\sigma_{\text{even=T}}$(a $\bowtie_{\text{a.n=b.k}}$ b))	
n	**lit**
0	zero
2	two

SELECT n, lit **FROM** a **JOIN** b **ON** a.n = b.k **WHERE** even = T

Figure 4: Illustration of the relational algebra

Listing 1: Output of EXPLAIN

```
Hash Join
  Hash Cond: (b.k = a.n)
  ->  Seq Scan on b
        Filter: (even IS TRUE)
  ->  Hash
        ->  Seq Scan on a
```

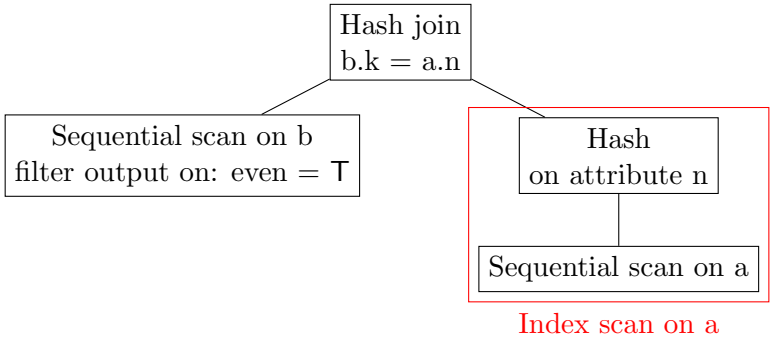

Figure 5: The query execution plan associated to figure 4

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

Figure 6: DataCert iterators, without axioms

The rewrite rules that transform relational algebra terms while preserving their interpretation are called *algebraic equivalences*. They have been proven correct by DataCert and are briefly described in Benzaken et al. [12].

The structure on which a QEP is built is known as the *physical algebra*. Low-level algorithms are operators of that algebra, while terms are either *iterators* or *indices*.

Iterators. A tuple iterator is an abstract list-like data container on which it is possible to perform the following operations:

- the *init* operation allocates whatever data structure in memory is necessary for iterating data: it is a required preliminary operation;
- the *next* operation produces a tuple (or tuple address) and alters the internal data structures of the iterator to reflect it;
- the *close* operation terminates the iterator's operation, by e.g. deallocating memory.

The QEP ultimately generates an iterator over all the tuples that were selected by the original request. The abstract representation of an iterator (called `Cursor`...) from [13] is partially reproduced in figure 6.

Cursors. The concept of iterator is inseparable from the notion of *cursor*. A cursor serves to remember previous operation of the iterator, and points to whatever corresponds to the current state of the iteration. Most iterators will use an adequate cursor internally. We have defined cursors for various data structures, namely association lists, hashtables and B+trees (see section 4).

Indices. Database indices are efficient implementations of the *finite map* interface, that can quickly tell the DBMS where a tuple is situated in memory.

Finite maps represent finite functional relations between keys and values, that is, relations that can be represented by a (partial) function, where keys map to a unique value. Let $\mathcal{M}(K, V)$ be the abstract type of finite maps storing keys of type K and values of type V. The minimal interface has two operations $\mathsf{lookup} : K \to \mathcal{M}(K,V) \to \mathsf{option}\ V$ and $\mathsf{insert} : K \to V \to \mathcal{M}(K,V) \to \mathcal{M}(K,V)$, where option is the usual option functor. It also has an *empty* map $m_\emptyset : \mathcal{M}(K,V)$. These operations must satisfy the following axioms:

$$\begin{aligned} \forall k, \quad \mathsf{lookup}\ k\ m_\emptyset &= \mathsf{None} \\ \forall k\ \forall k'\ \forall m, \quad \mathsf{lookup}\ k'\ (\mathsf{insert}\ k\ v\ m) &= \begin{cases} \mathsf{Some}\ v & \text{if} \quad k = k' \\ \mathsf{lookup}\ k'\ m & \text{otherwise.} \end{cases} \end{aligned} \tag{1}$$

lookup can be used to define an implementation-agnostic equivalence relation $\equiv$ on maps, stating that two maps hold exactly the same mappings:

$$\forall m\ \forall m', \quad m \equiv m' \quad \overset{\text{def}}{\Leftrightarrow} \quad \forall k, \quad \mathsf{lookup}\ k\ m = \mathsf{lookup}\ k\ m'$$

which then yields the properties:

$$\begin{aligned} & \mathsf{insert}\ k\ v'\ (\mathsf{insert}\ k\ v\ m) && \equiv && \mathsf{insert}\ k\ v'\ m \\ \text{and if } k \neq k', \quad & \mathsf{insert}\ k'\ v'\ (\mathsf{insert}\ k\ v\ m) && \equiv && \mathsf{insert}\ k\ v\ (\mathsf{insert}\ k'\ v'\ m)\ . \end{aligned}$$

Concatenated indices. Using C pointers, it is possible to accomodate any type for values by refering to their addresses in memory. However, known concrete implementations of indices usually handle atomic key types only, like integers (B+trees) or strings (tries). But what if we want to index data on a product type $K_1 \times K_2$?

The solution consists in implementing $\mathcal{M}(K_1 \times K_2, V)$ as $\mathcal{M}(K_1, \mathcal{M}(K_2, V))$. Given $(k_1, k_2) : K_1 \times K_2$ and $m : \mathcal{M}(K_1, \mathcal{M}(K_2, V)$, we implement the interface for $\mathcal{M}(K_1 \times K_2, V)$ by posing

$$\begin{aligned} \tilde{m}_\emptyset &:= m_\emptyset \\ \mathsf{lookup\_or\_empty}\ k\ m &:= \text{if}\ \ \mathsf{lookup}\ \ k\ m = \mathsf{Some}\ v\ \text{then}\ v\ \text{else}\ m_\emptyset \\ \widetilde{\mathsf{lookup}}\ (k_1, k_2)\ m &:= \mathsf{lookup}\ \ k_2\ (\mathsf{lookup\_or\_empty}\ k_1\ m) \\ \widetilde{\mathsf{insert}}\ (k_1, k_2)\ v\ m &:= \mathsf{insert}\ \ k_1\ (\mathsf{insert}\ \ k_2\ v\ (\mathsf{lookup\_or\_empty}\ k_1\ m)\ m\ . \end{aligned}$$

The proof that these definitions verify the axioms (1) is easy and, due to space constraints, left to the reader. *Concatenated indices* refer to such maps where the key type is composite; they are used to index tuples based on their projection on a list of attributes called a *subschema*.

Primary and secondary indices. *Entity integrity* is a database concept meaning that all the tuples in a database must be uniquely identifiable and capable of being located [5]. To maintain it, RDBMSs require relations to have a *primary key*, i.e. a subschema of the relation on which each tuple's projection is unique. While it is possible to define as primary key the whole schema if no two tuples are identical, that would be less performant than finding an adequate and minimal primary key.

RDBMSs systematically build an index on the primary key of a relation: it is called the *primary index* and defines the relation's content. That primary index is usually a B+tree when the primary key is composed of a single integer field, because they allow ordered iteration (see section 4).

RDBMSs can also build *secondary indices* indexing tuples on other attributes. Since tuples do not necessarily have unique projections on these attributes, a secondary index implements a finite map whose values are lists or iterators, or equivalently a kind of finite map interface with duplicates.

Iterator- and index-generating algorithms Here is a description of the database algorithms that I implemented.

- The *sequential scan* outputs an iterator over *all* the tuples stored in a relation R. It relies on the primary index of R and doesn't take any other input, thus lies at a leaf of the QEP.
- The *index scan* indexes tuples in a relation based on their projection on a given sub-schema. Its pseudo-code is in algorithm 1. The $\mathsf{insert}^{\ \mathsf{list}}$ function implements the insertion of an element $v : V$ at key $k : K$ in a "finite map with duplicates" of type $\mathcal{M}(K,\ \mathsf{list}\ V)$: if no mapping is found at key k, a singleton list $[v]$ is inserted. Otherwise, the current list l_k of values corresponding to key k is retrieved, v added to it, and the new list $v :: l_k$ is inserted back.
- The *index join* is a fast implementation of the $\bowtie$ abstract operator. When computing $R \bowtie R'$ and by analogy with the nested-loop algorithm, R will be called the *outer relation* and R' the *inner relation*. The index join uses a secondary index to lookup all tuples from the inner relation that can be joined with a given tuple from the outer relation. Its interface is in algorithm 2 and its implementation in

Data: A relation R and a subschema $(A_i)_{i=1}^n$ of R.
Result: A secondary index based on that subschema.
$m : \mathcal{M}(D_1 \times \cdots \times D_n,\ \mathsf{list}\ P) := m_\emptyset$
    where $\forall i$, D_i is the domain of A_i and P is the type of heap pointers;
it := the sequential scan iterator on R;
$\mathsf{it}.\mathsf{init}()$;
while it *is not empty* **do**
    (tuple) $t := \mathsf{it}.\mathsf{next}()$;
    $t_{\text{proj}} := \pi_{(A_i)_i}(t)$;
    $m := \mathsf{insert}^{\text{list}}\ t_{\text{proj}}\ \&t\ m$;
end
$\mathsf{it}.\mathsf{close}()$;
return m

Algorithm 1: The index scan algorithm

C is partially detailed in section 2. The interested reader can find detailed complexity analysis as well as descriptions of other efficient algorithms for joining relations in Mishra and Eich [20].

Data: An iterator it on relation R^{out} with schema $(A_i)_i$,
the schema $(B_j)_j$ of R^{in},
a list of pairs of foreign keys $(A_{i_k}, B_{j_k})_{k=1}^m$,
a secondary index $m : \mathcal{M}(D_1 \times \cdots \times D_m,\ \mathsf{list}\ P)$ on the subschema $(B_{j_k})_k$ of R^{in}
Result: An iterator on $R^{\text{out}} \bowtie_{\bigwedge_k (A_{i_k} = B_{j_k})} R^{\text{in}}$

Algorithm 2: The classic index (equi)join algorithm: interface

2 Implementation of some low-level algorithms in C

When I started my internship, DeepSpecDB already had some libraries implementing the main data structures used to index tuples' addresses in memory: B+trees [4] [9], hashtables, association lists. I developed database code that would use these libraries and implement the sequential scan, index scan and index join algorithms.

2.1 Object-oriented programming in C

Assume that in a query execution plan, an algorithm A takes input from iterators it_1 and it_2. The implementation of A only needs to have access to the *init*, *next* and *close* operations of it_1 and it_2. Moreover, we don't want the implementation of A to depend on what algorithm yields it_1 or it_2. This modularity is achieved through an *object-oriented* approach.

Object-oriented (OO) programming in C needs explicit method closures, as the language doesn't have built-in OO patterns. A C object can be implemented as a structure with a pointer to a *method table*, that stores function pointers to methods, and a pointer to an *environment* that stores private data. A *class interface* or *abstract class* or *signature* specifies the names of the methods and the types they must have. An object *class* describes a set of objects that have common methods with the same implementation. A class can implement an interface, in which case the interface's methods must be a subset of the class's methods. In our case, the abstract class or interface is "the iterators" and the classes are the physical iterator-generating algorithms.

The C type of environments can vary between different classes implementing the same interface, in order to account for different method implementation. That feature is achieved through pointer casts. The generic environment pointer has type **void***.

For example, instances of the sequential scan iterator class have an environment composed of a B+tree primary index pointing to a relation, and a cursor on that B+tree. The cursor is initialized to the first position by *init*, and moved to the next record by *next*. However, the methods and environment of the index join are completely different (see next section).

```
struct methods {
  void (*init) (void* env);
  const void* (*next) (void* env);
  void (*close) (void* env);
};
```

```
typedef struct iterator_t {
  struct methods *mtable;
  void* env;
} *iterator;
```

It is particularily unsafe to call methods directly, as there is no restriction on what the environment pointer actually points to. I therefore also defined convenient wrappers around an iterator's methods, like this `get_next` function, which guarantees that an iterator's `next` method is always called with the iterator's environment:

```
const void* get_next(iterator it) { return it->mtable->next(it->env); };
```

2.2 The index join: implementation

This section details the implementation of the `next` method of the index join iterator. That code implements the relational algebraic equijoin, modified to delete duplicate columns, as explained in section 1.2.1.

```
1  const void* index_join_next(void* env) {
2    struct index_join_env* e = (struct index_join_env*) env;
```

The defined function has the expected prototype for a `next` method. Line 2 casts the generic environment pointer into the index join's own environment pointer type. The index join environment holds essential data like the schema of both relations, the outer iterator, and the index on the inner relation.

```
3    while((e->current_inner == NULL || fifo_empty(e->current_inner))
4          && (e->current_outer = get_next(e->outer)) ) {
5      Key proj = get_projection(e->outer_attrs, e->current_outer, e->outer_join_attrs);
6      e->current_inner = (fifo*) index_lookup(e->ind_on_inner_sch, e->ind_on_inner, proj);
7    };
```

If the inner relation has no more tuple that matches the projection on join attributes of the current outer tuple, we query a new outer tuple from the outer iterator using `get_next`. In that case, we also retrieve the list (`fifo`) of inner tuples from the inner index that can be joined with the current outer tuple.

```
8    if(!e->current_outer) return NULL;
```

That **if** statement is triggered when the outer tuple iterator is now empty: the index join iterator is now also empty, thus the *next* method returns the null pointer.

Past line 8, we now have an outer tuple `e->current_outer` that has to be joined with *all* the inner tuples that are stored in the inner tuple list `e->current_inner`.

```
9    size_t outer_t_length = attribute_list_length(e->outer_attrs);
10   size_t inner_t_length = attribute_list_length(e->inner_attrs);
11   size_t join_length = attribute_list_length(e->inner_join_attrs);
```

Lines 9, 10 and 11 calculate the respective number of attributes in the outer tuples, the inner tuples, and the list of inner attributes that are going to be identified with the attributes from `outer_join_attrs` by the join algorithm. In the actual implementation, these numbers are calculated only once by *init*.

Crucial to understanding the rest of the code are the following statements:

(a) a single tuple is stored contiguously in memory, and

(b) independently of its domain, a field in a tuple occupies exactly **sizeof(void*)** bytes of memory, which is 8 on a 64-bits machine. This does *not* depend on whether the domain of that field is integer or string. Integer entries in our system are **unsigned long** and thus occupy 8 bytes and a string field is a pointer to the string characters, which is also 8 bytes. Note that unfortunately, these facts are not specified in the C standard, thus the code may not be portable.

```
12   void* new_t = malloc(sizeof(void*)*(outer_t_length + inner_t_length - join_length + 1));
13   memcpy(new_t, e->current_outer, sizeof(void*) * outer_t_length);
```

We allocate the memory space for the new joined tuple, calculating its size (in multiples of 8 bytes) using the length of the various attribute lists. Data from the outer tuple is entirely copied at the beginning of that new location.

```
14   const void* inner_t = fifo_get(e->current_inner)->data;
15   attribute_list l = e->inner_attrs;
```

We retrieve the next inner tuple in the list and create an alias for the list of the inner schema.

```
16   size_t n = 0;
17   for(; l != NULL; l=l->next) {
```

This loop will copy in the new joined tuple all fields in `inner_t` that are not already present.

```
18     size_t inner_t_ofs = get_offset(e->inner_attrs, l->id, l->domain);
```

`get_offset(attrs, attr_name, attr_domain)` returns the integer index of `attr_name` in the attribute list `attrs`, or a negative number if `attr_name` is not found. The `attr_domain` parameter allows to fail if `attr_name` exists in attrs, but with a different domain. An optimized version of the code would calculate the offsets only once at initialisation.

```
19     if(get_offset(e->inner_join_attrs, l->id, l->domain) < 0) {
20       memcpy((void*) (((size_t*) new_t) + outer_t_length + n),
21            (void*) ((size_t*) inner_t + inner_t_ofs), sizeof(size_t));
22       n++;
23     };
24   }
25   return new_t;
26 };
```

Finally, the index join is packaged into the iterator structure with its other methods whose details have been omitted here:

```
struct methods index_join_iterator_mtable =
     { &index_join_init, &index_join_next, &index_join_close };

iterator index_join([...]) {
  [...]
  struct index_join_env *env = malloc(sizeof(struct index_join_env)); if(!env) exit(1);
  [...initialize the environment structure...]
  iterator it = malloc(sizeof(struct iterator_t)); if(!it) exit(1);
  it->mtable = &index_join_iterator_mtable;
  it->env = (void*) env;
  return it;
}
```

2.3 Verification: perspective

Verification of object-oriented C code using Verifiable C is described by Appel et al. [8], chapter 29: *Dependently typed C programs*. The method that is presented will involve defining a generic Coq record type for the `iterator` abstract class that contains various definitions about the separation logic representation of iterators and proof of statements about these definitions. That Coq record defines what should be common to all implementations of iterators with respect to Verifiable C logic.

I have attempted at verifying that code, however I realized that it was too early to do so since the building blocks like indices were not done yet. Also, given that specifications are usually subtly modified as the proof is carried out, I will not reproduce here my attempts as they have not yet be proven to be fully correct. No Verifiable C verification has ever been attempted on such a large object-oriented code base.

3 A verified hashtable program

I achieved the verification of an implementation of *hashtables with external chaining*, that was used in our implementation of concatenated indices. The Verifiable C tutorial [7] guides students through the verification of a hashtable implementing a string multiset algebraic datatype by mapping string keys to their multiplicity in the multiset. The separation logic proof technique that is used is called *Magic wand as frame* and is the object of a recent paper [15] co-authored by my supervisor.

I will explain in this section how the Verifiable C verification of DeepSpecDB's integer hashtables led me to instantiate the magic-wand-as-frame technique in an interesting way.

3.1 Purpose and C code

As explained in section 1.2.2, we need to be able to build various kinds of indices for the join algorithm. Each of our attribute domains - strings or integers for now - has its own implementation of a key-pointer store: string association lists and integer hashtables, respectively. A concatenated index is implemented as an index pointing to other indices.

The hashtable data structure consist of an array of pointers to linked lists, called buckets. The bucket at array position i contains any entry whose integer key k verifies $k \bmod N = i$, where N stands for the total number of buckets in the array. *External chaining* means that the hashtable handles collisions of hashes of different keys by growing the linked list buckets. We use a large and prime N, namely 65599, to distribute entries evenly on average and guarantee fast lookups.

The insertion operation has a main procedure `inthash_insert` and an auxiliary function `inthash_insert_list` that are both reproduced in figure 7.

The `inthash_insert_list` function returns a pointer to the adequate cell in a bucket, by creating a new cell if needed. The pointed cell has to be updated with the new value by the `inthash_insert` function, which itself returns the old value.

3.2 Separation logic representation

Notice that the `inthash_insert_list` function can modify the pointer in the bucket array, in the case the bucket is empty, hence the use of a pointer to a pointer in the arguments of the function. The paper and the Verifiable C tutorial describe such a situation using a **listboxrep** predicate, along with

```
struct icell {
  size_t key;
  void *value;
  struct icell *next;
};
```

```
struct inthash {
  struct icell *buckets[N];
};
```

```
1  struct icell *inthash_insert_list (struct icell **r0, size_t key) {
2    struct icell *p, **r;
3    for(r=r0; ; r=&p->next) {
4      p = *r;
5      if (!p) {
6        p = new_icell(key,NULL,NULL);
7        *r = p;
8        return p;
9      }
10     if (p->key==key) {
11       return p;
12     }
13   }
14 }
15
16 void* inthash_insert (inthash_t p, size_t key, void *value) {
17   struct icell *q;
18   void *old;
19   q=inthash_insert_list (& p->buckets[key%N], key);
20   old=q->value;
21   q->value=value;
22   return old;
23 }
```

Figure 7: Hashtable definition and insertion

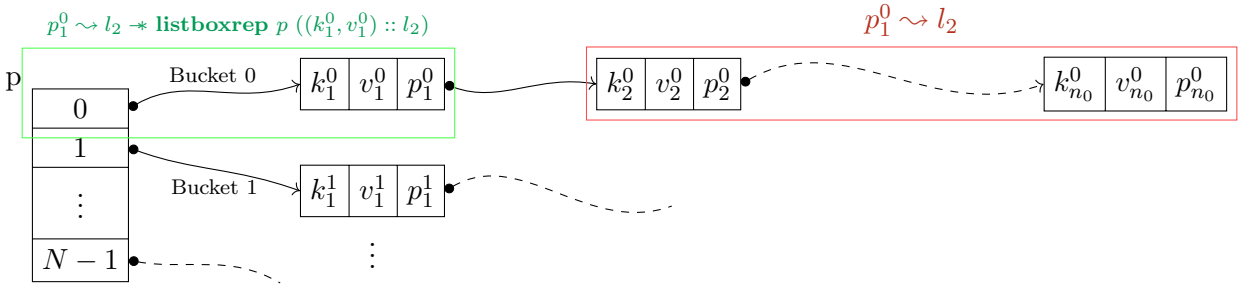

Figure 8: Hashtable illustration and separation logic predicates, where $l_2 := [(k_2^0, v_2^0), \ldots, (k_{n_0}^0, v_{n_0}^0)]$ and p is the address of the first bucket pointer.

the standard separation logic representation predicate for linked lists:

$$\begin{array}{ll} p \rightsquigarrow \text{nil} & := \ p = \text{null} \\ p \rightsquigarrow (k, v) :: tl & := \ p.\mathsf{key} \mapsto k * p.\mathsf{value} \mapsto v * \exists q, p.\mathsf{next} \mapsto q * q \rightsquigarrow tl \end{array}$$

$$\textbf{listboxrep}\ l\ p \quad := \exists q, p \mapsto q * q \rightsquigarrow l$$

The notations are as follows:

- $*$ is the separating conjunction: a heap m satisfying $A * B$ (written $m \models A * B$) can be split into two disjoint parts m_1 and m_2 such that $m_1 \models A$ and $m_2 \models B$;
- $p \mapsto v$ means that value v is stored at memory address p on the heap;
- nil is the empty list and :: is the constructor of functional lists - in this case, association lists or lists of pairs;
- finally, null is the null pointer.

The separating implication will be denoted $A \mathrel{-\!\!*} B$. By definition, $m \models A \mathrel{-\!\!*} B$ if and only iff for all m' disjoint from m such that $m' \models A$, we have $m \oplus m' \models B$, where $\oplus$ is heap disjoint union. Intuitively, $A \mathrel{-\!\!*} B$ describes a heap A where a part B is missing.

The magic-wand-as-frame paper advocates the use of that separating implication, or *magic wand*, to describe partial data structures that appear on the heap when a recursive data structure (e.g. binary tree, linked list) is traversed. The magic wand then allows for elegant loop invariants that *do not* require additional inductive predicates for partial structures to be defined (e.g. list segments or partial trees). The paper shows that this technique can be used when the traversal doesn't modify the heap (e.g. for lookups) but also and more interestingly *when it does*, like here. In the latter case, the wand conjunct has to use a universal quantifier over the remaining sub-structure, to be modified by the traversal. More details are given below.

In the body of `inthash_insert_list` comes a loop (lines 3 to 13) in which the *pointer value* stored at address `r` goes through the bucket until reaching the end of the linked list or the current mapping for key k. With respect to the magic-wand-as-frame technique, I used the following loop invariant:

$$\begin{array}{r} \exists (l_1, l_2),\ l = l_1 l_2\ \wedge\ [\![\mathtt{r}]\!] = r\ \wedge\ [\![\mathtt{r_0}]\!] = r_0\ \wedge\ [\![\mathtt{k}]\!] = k\ *\ \textbf{listboxrep}\ r\ l_2\ * \\ \forall l_2',\ \textbf{listboxrep}\ r\ l_2'\ \mathrel{-\!\!*}\ \textbf{listboxrep}\ r_0\ (l_1 l_2') \end{array}$$

The concatenation of two lists l_1 and l_2 is written $l_1 l_2$ above. The universal quantification on the sublist whose starting address is stored at r before the wand means that using that loop invariant, it is possible to alter that list when needed. In the case of `inthash_insert_list`, that happens when a new cell has to be added at the end, because the key being searched wasn't found. In that case, the new cell will be given the NULL mapping. That value will be modified by the `inthash_insert` function.

When specifying `inthash_insert_list`, it is necessary to include the fact that the returned value is pointing to the list cell that needs to be modified by `inthash_insert`. I thus came up with the following specification:

$$\begin{array}{rl} \text{(precondition)} & \{[\![\mathtt{p}]\!] = p \wedge [\![\mathtt{k}]\!] = k \wedge \textbf{listboxrep}\ l\ p\} \\ \text{(statement)} & \quad p_{\text{ret}} = \texttt{inthash\_insert\_list}(\mathtt{p}, \mathtt{k}) \\ \text{(postcondition)} & \{\exists (r, v_0, tl),\ r \mapsto p_{\text{ret}} * p_{\text{ret}} \rightsquigarrow (k, v_0) :: tl\ * \\ & \qquad \forall v,\ r \mapsto p_{\text{ret}} * p_{\text{ret}} \rightsquigarrow (k, v) :: tl \mathrel{-\!\!*} \textbf{listboxrep}\ (\textbf{add}\ k\ v\ l)\ p\} \end{array}$$

Here, **add** is the insertion on functional association lists. To have a sufficiently strong postcondition, it is *not possible* to use the **listboxrep** predicate for the two first conjuncts and before the wand in the postcondition. If we do, because of the existential quantification in **listboxrep** on the value stored at r, we lose the information that the returned value p_{ret} points to a sublist whose head key is the inserted key. That's why, contrary to the paper's examples, I had to unfold the definition of **listboxrep**.

Notice how adequate and specific to the problem that specification is. Once `inthash_insert_list` is proven to conform to that specification, the proof of `inthash_insert` - with a natural specification matching the new hashtable in the heap with the same hashtable where key k now maps to value v - goes very smoothly, because most of the work is done in that auxiliary proof.

Finally, the C code could have been modified in the following way: pass the new value to `inthash_insert_list`, and make it perform the update and return the old value. In that case, we wouldn't have had to give a magic-wand postcondition for `inthash_insert_list`, which may have been a little simpler. However, it is interesting to see that the magic-wand-as-frame technique can be adapted to various situations like this piece of C code.

4 A verified B+tree functional implementation

B+trees are a specific kind of multiway search trees that store ordered mappings of a key to an associated value. B+trees are derived from B-trees, which were originally described by Bayer and McCreight [10]. B+trees have been designed to handle massive amounts of data while allowing for the shortest lookup time. Such trees are typically very wide and shallow, so that the number of nodes fetched from cold areas of cache or main memory on a path from the root to a leaf is small. The number of children of a node is also chosen so as to minimize cache misses, that is, so that one node of a B+tree is the same size as the block size of one layer of the memory hierarchy (disk or cache).

This section describes the B+tree data structures and its invariants, then exposes some challenges I faced during formalization in Coq.

4.1 Presentation of B+trees

Informally, B+tree nodes have the following structure. A *leaf node* will contain a number of *entries* which are key-value pairs and solely constitute the information contained in the abstract key-value store. An *internal node* contains a *zero* (or *leftmost*) *pointer* and a list of key-child pointer pairs - also called entries. The entry keys guide the lookup operation towards the right mapping in the appropriate leaf. The lookup top-to-bottom traversal is similar to that of any binary search tree - except that the number of children is greater. Contrary to B-trees, B+tree internal nodes do not carry any mapping information: their only purpose is to make lookups fast using intermediate keys.

Formally, the inductive definition of the base type of the B+trees is the following:

$$t \quad = \quad T(t_0, (k_i, t_i)_{i=1}^{n_t}) \quad | \quad L((k_i, v_i)_{i=1}^{n_t})$$

where $t, t_0, t_1, \ldots, t_{n_t}$ are trees, the k_i's are keys, the v_i's are values, a pair (k, x) where k is a key and x is either a tree or a value is called an *entry*, and the number n_t of entries is tree-dependent. The words *tree* and *node* will be used interchangeably in the rest of the explanations. The constructor T refers to internal nodes, while L refers to leaves.

Along with that definition, we will use the usual notions of *child*, *tree path* and *path length*. An internal node $T(t_0, (k_i, t_i)_{i=1}^{n_t})$ has $n_t + 1$ children, which are trees $t_0, t_1, \ldots, t_{n_t}$. That is, the leftmost pointer is a child, and so are the trees in the entries. Leaf nodes have no child. If t_{child} is a child of t_{parent}, we write $\mathsf{child}\ t_{\text{child}}\ t_{\text{parent}}$. A path will be an ordered collection of distinct nodes $(t_0, t_1, \ldots, t_d)$ such that $\forall i \in \{0, \ldots, d-1\}$, t_{i+1} is a child of t_i. Such a path is said to have length d, which is the number of edges it is composed of.

A node t_{child} is said to be a *subnode* of another node t_{parent} if there is a path from t_{parent} to t_{child}. That defines a predicate and we write $\mathsf{subnode}\ t_{\text{child}}\ t_{\text{parent}}$. The $\mathsf{subnode}$ relation is the reflexive transitive closure of the child relation.

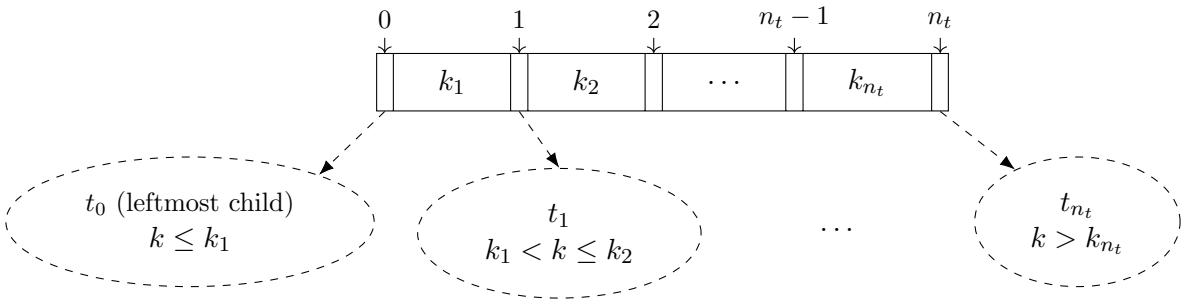

Figure 9: A B+tree internal node. The numbers above indicate child positions used in B+tree cursors.

Given such a multiway tree t and a positive integer $m \in \mathbb{N}_{>0}$, we say that t is a B+tree of order m if it satisfies the following invariants.

1. t is *balanced* in the sense that *all paths from t to one of its leaves have the same length.* That length is called the *depth* of the B+tree.

   That property guarantees that looking up any key in the tree will have the same time complexity. It is an even stricter balancing invariant than the one of *AVL trees*, which are binary search trees allowing pairs of subtrees to have a height difference of 1. This invariant is concomitant to the way an insertion is performed: as we will see, during B±tree insertion, a new node is always a sibling of another preexisting node, or a new root; in other words, B±trees grow in height at the root, not the leaves. That behavior is inherited from 2-3 trees, of which B-trees are a generalization.

2. t is *well-sized*, that is t has at most $2 * m$ entries and any strict subnode of t has between m and $2 * m$ entries. If t is not a leaf, then t must also have at least 1 entry: no internal node has an empty list of entries.

   That invariant guarantees that the complete B±tree will have a number of nodes that is not too high compared to the number of mappings it contains. The interest of that invariant is to rule out nearly-empty nodes. In the early days of databases, all the data was stored on slow discs, and a B+tree node's entries would be stored in a dedicated sector or block. Having nodes with few entries would increase disk usage as well as the total depth of the tree, which decreases cache locality of the program and makes lookups slower. The same reasoning applies today to RAM blocks and/or cache lines.

3. t is a *search tree*. That property is crucial not for performance, but for proper operation.

   For a leaf node $L((k_i, v_i)_{i=1}^{n_t})$, we demand that the list of keys $k_1, \ldots, k_{n_t}$ be sorted, with respect to whatever strict total order the type of keys comes equipped with. A B+tree doesn't allow duplicate keys, however an implementation allowing them seems feasible.

   For an internal node $t = T(t_0, (k_i, t_i)_{i=1}^{n_t})$, we require that

   (a) the list of keys $k_1, \ldots, k_{n_t}$ is sorted;

   (b) all the children of t are search trees;

   (c) all keys k in t_0 are such that $k \leq k_0$ (remember that, if the tree is well-sized, then $n_t > 0$, so k_0 exists);

   (d) for any $i \in \{1, \ldots, n_t - 1\}$, the keys k in t_i are such that $k_i < k \leq k_{i+1}$; and

   (e) the keys k in t_{n_t} are such that $k > k_{n_t}$.

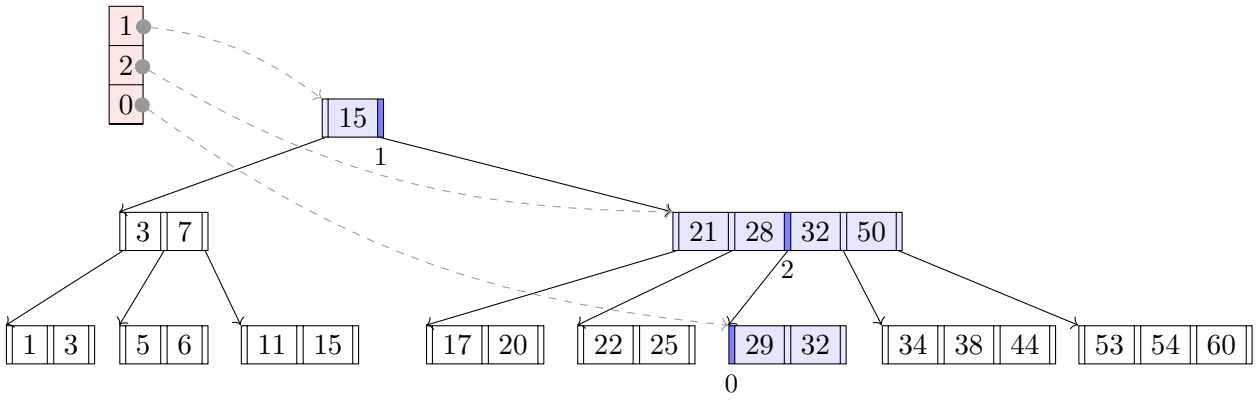

Figure 10: A sample B+tree of depth 2 showing the "search tree" property, with parameter $m = 2$. The cursor in light red points to key 29. The nodes on the path designated by the cursor are in light blue. The numbers in the cursor represent the position of the adequate child or record in the node that is pointed to. The corresponding pointers in the B+tree are highlighted in blue.

4.2 Coq model and proofs

To my knowledge, the only proof assistant formalization of B+trees in the literature is described by Malecha et al. [18]. Their datatype for B+trees includes a `nat` height parameter, which allows Coq to check the termination of recursion easily. They define a tree traversal that is supposed to serve for the implementation of different B+tree operations. However, they only really use it for insertion, which is the only operation they define along with lookup. I didn't define or verify the removal operation either, which is expected to be quite lengthy. However, my formalization is made to match the *cursored* operations we use in our C code.

A B+tree cursor is a path from the root to a leaf record, that is kept in memory by the B+tree C program. They can be found in the B+tree implementation of SQLite [5]. They allow speedups on large trees when consecutive lookups and insertions are made *locally* in the tree. Moreover, they are an alternative to cross-leaf pointers used in most implementations of B+trees [18] [17] for iterating over all sorted mappings contained in the tree. A cursor is presented in figure 10.

Using a cursor, the lookup and insertion operations do not have to traverse the tree from the root. The leaf where the key is found is reached from the current position of the cursor. If a sequence of keys is inserted in a sorted or mostly-sorted order, the cursor can provide a noticeable speedup, because the cursor would move very little, and would trigger fewer cache misses compared to cursorless insertions or lookups, that would need to traverse the whole tree from top to bottom.

My formalization uses a Coq list to describe the entries in a B+tree node, which feels like the most natural way to define multiway search trees. It takes the form of a Coq functor, and is inspired from the `FMapAVL` file [1] from Coq's standard library, which implements the finite map interface in `FMapInterface` using fully verified AVL trees. However, that part of the standard library is unfortunately deprecated and doesn't seem to be maintained so I chose not to rely on it.

```
Module Type BPlusTreeParams.
  Module InfoTyp.
    Parameter t: Type.
  End InfoTyp.
  Parameter min_fanout: Z.
  Axiom min_fanout_pos: min_fanout > 0.
End BPlusTreeParams.

Module Raw (X: Orders.UsualOrderedTypeFull') (Params: BPlusTreeParams).
```

```
  Notation key := X.t.
  Definition entries (elt: Type): Type := list (key * elt).

  Inductive node: Type :=
  | leaf (records: entries elt) (info: Info.t): node
  | internal (ptr0: node) (children: entries node) (info: Info.t): node.

  [...]
End Raw.
```

Each node stores an `Info.t` informative value, with no constraint. The `min_fanout` parameter corresponds to the m B+tree parameter above. Keys need to be equipped with a total order. I have investigated the use of any equivalence relation for equality of keys, however that was not needed for our purpose and required a lot more work to prove that the non-Leibniz equality was respectful of all other definitions, which is needed for setoid rewriting. Thus the key type is an instance of `UsualOrderedType` and not `OrderedType`.

A *nested inductive type* occurs in the `children` argument of the `internal` constructor, as it is a list whose element type contains the inductive type `node` being defined. Coq can handle nested inductive types, however I had to convince the termination checker that my inductive definitions on `node` were terminating, either by structural recursion or using well-founded relations. An alternative definition uses mutual inductive types to redefine a list of children for `node`, isomorphic to the Coq type `list`, but is very impractical given that all definitions and lemmas about lists cannot be reused.

For such nested inductive types, the induction principle automatically defined by Coq is too weak. A proper induction principle can however be proved. Different choices are possible. I chose the `Forall` predicate for the induction hypothesis:

```
Lemma node_ind (P: node → Prop)
  (hleaf: ∀ records info, P (leaf records info))
  (hinternal: ∀ ptr0 children info,
      P ptr0 → Forall P (map snd children) → P (internal ptr0 children info)):
∀ n, P n.
```

Two methods are documented for recursive definitions on such nested inductive types:

Well-founded relations. The first method is to use a well-founded relation, or to use a measure and a predefined relation (e.g. less-than on natural numbers). My take on that was inspired by what is done in the Equations package [19]. Equations has tactics and a command `Derive Subterm` that try to define automatically a *subterm* relation for recursive inductive types and prove its well-foundedness. In my case, I defined the following:

```
Inductive child: relation node :=
| child_ptr0: ∀ ptr0 records info,
    child ptr0 (internal ptr0 records info)
| child_entry: ∀ n ptr0 children info,
    In n (map snd children) →
    child n (internal ptr0 children info).

Lemma child_wf: well_founded child.
```

I used that technique to define the `get_cursor` function which, given a key k and a B+tree, returns a list of nodes and node indices on the path from the root to where the B+tree stores (or would be storing) k. That "list of indices and nodes" cursor type is the reason why I didn't encode the height of a tree (and its balancing property) inside the `node` datatype itself.

```
Function get_cursor (k: key) (root: node) {wf child root}: list (Z * node) :=
match root with
| leaf records info ⇒ [(find_index k records, root)]
| internal ptr0 children info ⇒
    let i := find_index k children in
    (i, root):: get_cursor k (if i =dec 0 then ptr0
                              else Znth (i − 1) (map snd children))
end.
Proof. [...]   Defined.
```

The `find_index` function returns a number indicating which child of an internal node should contain a given key, or in the case of a leaf, at which position in the entries list a mapping with a given key is or should be inserted.

Structural recursion and nested fixpoints. Another method is to use basic structural recursion. But Coq won't accept all fixpoint definitions if the structurally smaller argument is a `node`, even though most of the time termination is obvious to the user.

The following *flatten* function, which transforms a B+tree into an ordered association list, works straight out-of-the-box:

```
Fixpoint flatten (n: node) {struct n}: entries elt :=
match n with
| leaf records _ ⇒ records
| internal ptr0 children _ ⇒ flatten ptr0 ++ flat_map (flatten ∘ snd) children
end.
```

However, to define the `search_tree` predicate formally described in section 4.1, one needs to define the treatment of the list of children using a *nested fixpoint* for Coq to understand that the recursive call happens on a structurally decreasing argument. That technique is described in various sources like in Chlipala [16], Bertot and Castran [14] and the Coq-Club mailing list archives.

```
Fixpoint search_tree (min max: key) (n: node) {struct n}: Prop :=
match n with
| leaf records info ⇒ Sorted (<key) (map fst records) ∧
    min ≤key max ∧
    Forall (fun k ⇒ min ≤key k ∧ k ≤key max) (map fst records)
| internal ptr0 children info ⇒ ∃ max_ptr0,
    search_tree min max_ptr0 ptr0 ∧
    (fix st_children l min: Prop :=
    match l with
    | nil ⇒ min ≤key max
    | (k, n):: tl ⇒
        min ≤key k ∧
        ∃ min_n max_n,
          k <key min_n ∧
          search_tree min_n max_n n ∧
          st_children tl max_n
        end) children max_ptr0
    end.
```

The meaning of `search_tree` m M n is the following: n is a search tree, and all its keys k are such that $m \leq_{\text{key}} k \leq_{\text{key}} M$.

Among the properties proved about `search_tree` is the sortedness (with respect to key order) of the association list obtained by flattening a node satisfying `search_tree`. I also defined the *insertion with cursor* B+tree operation. Its code is not reproduced here for conciseness. The insertion uses a list of

nodes and integer positions output by `get_cursor`, and takes a key and a value to insert in the B+tree. Starting from the leaf pointed to by the cursor, it inserts the new entry at the position given by the integer index. If the node becomes too filled to satisfy the well-sized invariant, it is split in 2 distinct nodes, that need to be updated or inserted in the parent. The procedure goes up the tree along the cursor, updating and inserting a new child as needed: if internal nodes become too large, they are also split and generate a new node to insert above them. The tree grows with a new root if the current root has itself to be split.

Finally, the main results of my development[3] is the conservation of the different B+tree invariants after insertion. For instance, I proved the following statement (400 lines of code):

```
Theorem insert_search_tree: ∀ root m M k0 e,
    search_tree m M root → search_tree (key_min m k0) (key_max M k0) (insert k0 e root).
```

`key_min` and `key_max` being defined using the functor `GenericMinMax` from the standard library.

5 Conclusion

My work has spanned several facets of the design of verified low-level database algorithms. While formal methods are nowadays commonly used for ensuring that critical systems comply with their specification, they have surprinsingly seldom been applied to data-centric systems. Therefore, an in-depth study of the topic was a required first step. The implementation of the standard algorithms led me to use libraries that had been incompletely verified, or not verified at all. I contributed to their Verifiable C verification (hashtables) as well as their Coq formalization and verification (B+trees).

Overall, I became very experienced with Coq, I gained a deep understanding of the operation of RDBMSs, and I could put my knowledge of separation logic in practice using Verifiable C.

[3] Available in the DeepSpecDB repository at `www.github.com/PrincetonUniversity/DeepSpecDB/blob/master/verif/btrees/btrees_functional.v`

References

[1] The Coq proof assistant: Library Coq.FSets.FMapAVL. `https://coq.inria.fr/library/Coq.FSets.FMapAVL.html`. Accessed: 2019-08-15.

[2] The DataCert project: Coq deep specification of privacy aware data integration. `http://datacert.lri.fr`. Accessed: 2019-08-15.

[3] The DataCert project: Roadmap. `http://datacert.lri.fr/roadmap.html`. Accessed: 2019-08-15.

[4] Oluwatosin Victor Adewale. Implementing a high-performance key-value store using a trie of B+Trees with cursors. Master's thesis, Princeton University, 2018.

[5] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Apress, Berkeley, CA, USA, 2nd edition, 2010.

[6] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq : A verified compiler for Coq. 2016.

[7] Andrew Appel. Software foundations - volume 5beta: Verifiable C. `https://www.cs.princeton.edu/~appel/vc/`. Accessed: 2019-08-15.

[8] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[9] Aurèle Barrière. VST verification of B+Trees with cursors. M1 internship report.

[10] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Sep 1972.

[11] Véronique Benzaken and Évelyne Contejean. A Coq mechanised formal semantics for realistic SQL queries - Formally reconciling SQL and bag relational algebra. working paper or preprint, July 2018.

[12] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A Coq formalization of the relational data model. In Zhong Shao, editor, *Programming Languages and Systems*, pages 189–208, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[13] Véronique Benzaken, Évelyne Contejean, Chantal Keller, and Eunice Martins. A Coq formalisation of SQL's execution engines. In *ITP 2018 - International Conference on Interactive Theorem Proving*, volume 10895 of *Lecture Notes in Computer Science*, pages 88–107, Oxford, United Kingdom, July 2018. Springer.

[14] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[15] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. Proof pearl : Magic wand as frame. 2018.

[16] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[17] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[18] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM.

[19] Cyprien Mangin and Matthieu Sozeau. Equations reloaded. working paper or preprint, July 2018.

[20] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, March 1992.

[21] The PostgreSQL Global Development Group. *PostgreSQL 11.5 Documentation*, 2019.