VST Verification of B+Trees with Cursors

Aurèle Barrière

École Normale Supérieure de Rennes, France
`aurele.barriere@ens-rennes.fr`

Supervisor:
Andrew Appel
Princeton University

March 1st, 2018 - June 29th, 2018

Abstract. The DeepSpecDB project aims to define, specify and verify a high-performance concurrent in-memory database system. Based on MassTree, it uses B+Trees, a well-studied key-value data structure. Our sequential B+Trees library uses cursors, introduced in the database engine SQLite. Such cursors reduce the complexity of operations when dealing with partially sorted data. We define a Coq formal model for such trees, then use it to specify and prove the correctness of the C implementation using the Verified Software Toolchain.

Keywords: B+Tree, Cursor, VST, Verification, Coq, C

1 Introduction

With memory sizes increasing and prices dropping, the assumption that most of the values of a database system must reside on disk no longer holds. This resulted in the emergence of several *Main Memory Database Systems* [18], where most (or all) of the data is in the memory. This leads to much faster database operations, as reading from the volatile memory is typically orders of magnitude faster than the disk, at the cost of a few changes in architecture design. For instance, the size of memory blocks should be adapted to fit into cache lines instead of disk sectors. Examples include MassTree [23], VoltDB [8], MemSQL [3], Hekaton, SAP HANA and others.

MassTree [23] is an example of such Main-Memory Database System. It stores values, indexed by keys, directly in the memory. It uses a combination of the well-studied B+Trees (a variation of the BTrees introduced in [14]) and Tries data structures. MassTree's performance is similar to MemCached, and better than VoltDB, MongoDB and Redis, other high-performance storage systems.

Formal verification of C programs has been the focus of many recent works. In particular, CompCert [4, 21] is a fully verified C compiler in Coq [5], which formally defines C and assembly semantics and proves that the source and compiled programs have equivalent behaviors. This allows for program verification at the source level, as the compiled program is guaranteed to run as specified by the source code. Then, the Verified Software Toolchain (VST) [7] allows you to write specifications for C programs and formally verify in Coq that they are respected (using the same C semantics as CompCert). VST is itself proved sound in Coq with regards to CompCert's semantics. VST has been used to prove correctness of many C programs, including cryptographic primitives such as SHA-256 [13] or OpenSSL's HMAC [16].

This paper focuses on the implementation, specification and verification of a B+Trees with Cursors Library. As of today, this library only deals with sequential operations. Ideally, our data structure should allow a concurrent usage, but we believe that the formal verification of a sequential program is a mandatory first step towards the verification of a concurrent one. Given a formal specification of an abstract key-value data structure with cursors and a first version of a B+Tree with cursors implementation, the work presented here consisted in rewriting the C code to comply with the specification, then prove it correct using VST.

Every C function of the library has been proved correct with regards to a formal specification. The verification of the B+Tree Library has allowed us to find several bugs in the original implementation (see Section 6).

We first present the B+Tree Structure with cursors and its implementation (Section 3) We then define an equivalent formal model, that is used for the verification (Section 4). We then present in Section 5 the VST verification of our implementation. We present in Section 6 some of the bugs we encountered and fixed while verifying the library.

2 Related Works

This verification work is part of the DeepSpecDB project [2]. Its goal is to provide a verified library for a high-performance concurrent Main-Memory database. Inspired by MassTree, this database system also uses a combination of B+Trees and Tries structures.

As part of the DeepSpecDB project, a formal Coq specification of abstract relations with cursors has been defined in [24]. A C implementation of a sequential version of the database has also been done in [9]. This implementation contains a first version of a library for B+Trees with cursors, a library for border nodes [23] and a library for Tries. The work presented in this paper is the first verification step: it includes modifying the B+Trees library and proving it correct with regards to the abstract specification.

Previous work has dealt with the verification of tree-like structures. Verified Functional Algorithms [12] includes a Coq formal model for Binary-Search Trees and its verification. Red-Black Trees have also been verified in [11]. [19] provides a Coq correctness proof of the AVL Trees used in the Set Module of the OCaml standard library. VST's examples [7] include the VST verification of a C implementation of Binary-Search Trees. Recent work has been made to formalize SQL semantics [15].

Even though the current version is sequential, we believe that implementing concurrency would be a feasible next step. Our verification work could then be used as a basis for the new concurrent library. Indeed, VST can also be used to prove correctness of concurrent C programs [22].

3 B+Trees with Cursors Library

3.1 B+Trees

The first part of our work consisted in modifying a B+Trees with cursors library. B+Trees are ordered and self-balanced. This allows for fast access to the data (located at the leaves), as it suffices to go down the tree using the keys in the nodes to find the next one. B+Trees have been well studied [25] and implemented numerous times. A B+Tree example can be seen **Fig.** 1. The keys in the leaves point to a record (indicated in the figure by a * next to the key). The fanout value in this example is 4 (the maximum number of keys in each node). Every internal node has $n + 1$ children, where n is its number of keys. Traditionally, B+Trees implement cross-links between leaves, meaning that each leaf node points to the previous and the next one. This allows for range queries as one can always find the next record. However, because our implementation uses cursors (see below), these links are not needed.

The available operations on a B+Tree typically include inserting a record (associated with a key), deleting a record, accessing the record associated to a given key (if it has been inserted) and accessing the records corresponding to a range of keys. We briefly describe some of these operations. More details can be found in [25].

Insertion Takes a key and a record. If the key already exists in the B+Tree, its associated record should be updated to the given one. Otherwise, the record is added to the leaves. If it is inserted in a full leaf node, this node should be split into two. Then, the middle key is copied into the parent node to point to the new one. When inserting this new key, it is possible that the parent node was also full. In that case, we keep on recursively splitting nodes until some parent can accept a new entry, or the root has to be split (thus creating a new level in the tree). This algorithm keeps the tree balanced. The complexity of inserting a new record is $\mathcal{O}(\log_f(n))$, where f is the fanout of the tree and n the number of records. Indeed, the number of operations (key comparisons and node splittings) is linear in the depth of the tree.

For instance, a record with the key 4 should be inserted in **Fig.** 1 in the first leaf node (containing keys 0, 1, 2 and 3). Because this node is full, it should be split into two: one containing 0, 1, 2, and the other containing keys 3 and 4. Then, key 2 should be inserted into the parent node (with keys 5, 9, 12, 15). Because this node is full too, it should be split. Because internal nodes do not contain records, the middle key can simply be pushed to the root without being copied. The resulting B+Tree, after insertion, can be seen in the appendix **Fig** 6.

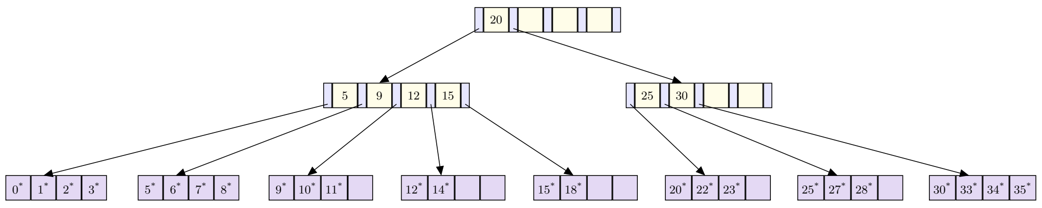


Fig. 1. A B+Tree

Accessing a Record Takes a key as input. Because the tree is ordered, it suffices to go down from the root. At each node, the values of the keys indicate which child to go to at the next step. The complexity is $\mathcal{O}(\log_f(n))$, where f is the fanout of the tree and n the number of records.

3.2 Cursors

On numerous occasions, inserting or accessing data can be done on partially sorted keys. In this case, the operations will target and affect the same part of the B+Tree. But this locality isn't exploited, as the functions always start from the root. Cursors aim to exploit the locality of operations on close keys, by remembering the last position where the tree was modified or accessed. Then for instance, to look for a new key, the function can start from the last accessed leaf. If the searched key is in the same node, then the function has constant time complexity. Otherwise, it should go up to the node's parent before going down again. If the searched key is close, this should significantly reduce the number of node accesses. Abstractly, cursors point to a position in the ordered list of key-value pairs represented by a B+Tree. Their main purpose is to allow fast operations that go through a B+Tree sequentially, such as range queries.

A cursor is implemented as an array of pairs. Each pair contains a pointer to a node and an index for that node. The first node of the cursor should be the root of the B+Tree. Then the following nodes describe a path from the root to an entry in a leaf node. An example of cursor can be found in **Fig.** 2. A cursor's length should always be equal to the depth of the B+Tree it refers to, thus pointing to a record. To the best of our knowledge, cursors of this kind have been used only in the B+Trees implementation of SQLite [6].

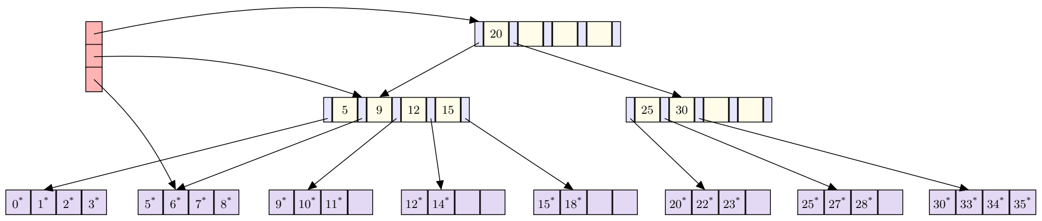


Fig. 2. A B+Tree with a cursor pointing to key 6

With a cursor, basic functions (insertion, accessing a record) do not start at the root anymore, but from the leaf node that the cursor is pointing to. For instance, when accessing the record for a given key k, if k is in the same leaf node as the cursor, then we can fetch the associated record in constant-time. Whenever k is not in the same leaf node, we need to go up in the B+Tree using the previous levels of the cursors, until we reach a parent node of the desired key. We can then go down the B+Tree until k is found. For instance, if we search for key 14 in **Fig.** 2, we can first see that 14 is not in the leaf node pointed to by the cursor (containing only keys from 5 to 8). We then go up in the B+Tree, using the cursor, to the internal node containing 5, 9, 12 and 15. Because it contains keys less and greater than 14, we know that this node is a parent node for 14. Finally, we go down to the fourth child, containing keys 12 and 14, and we can get the desired record. This new algorithm is still $\mathcal{O}(\log_f(n))$ in the worst-case (if we need to go up to the root of the tree, then go down to the leaves), but is faster when looking for close keys. In particular, accessing the record of each key sequentially has amortized constant time complexity [9].

Inserting a new record has a similar complexity. To perform range queries, we need to introduce a new function, *movetonext*, that moves the cursor to the next record in a B+Tree.

3.3 Implementation

We modified DeepSpecDB's implementation of B+Trees with cursors. The C types are the following:

```
typedef struct Relation {
    struct BtNode* root;
    size_t numRecords;
    int depth;
} Relation;

union Child_or_Record {
    BtNode* child;
    const void* record;
};

struct Entry {
    Key key;
    Child_or_Record ptr;
};
```

```
struct BtNode {
    Bool isLeaf;
    Bool First;
    Bool Last;
    int numKeys;
    BtNode* ptr0;
    Entry entries[FANOUT];
};

struct Cursor {
    Relation* relation;
    int level;
    int ancestorsIdx[MAX_TREE_DEPTH];
    BtNode* ancestors[MAX_TREE_DEPTH];
};
```

A node contains an array of entries. Each of these entries contains the pointer to the associated child (for internal nodes) or the associated record (for leaf nodes). Because internal nodes need one more child than they have entries, the node also contains `ptr0` which holds the pointer to the first child (or `NULL` for a leaf node). Nodes also contain booleans that indicate if the node is the first or the last of its level in the B+Tree. This is used to speed up some functions. Finally, a cursor has an array of ancestors (the nodes, from the root to the leaf node pointed to) and an array of indexes locating the next child at each level.

This library contains 27 functions, all of which are verified in VST (see Section 5). Deletion hasn't been implemented yet. The list of all functions that can be used by a client of the library can be seen in the appendix, **Fig.** 8.

4 A Formal Model for B+Trees with Cursors

In order to prove the correctness of our B+Trees library, we first need a formal model, in Coq, for each type and function. This model will be used to specify each C function with VST. Then, the correctness can be proved by showing that the functions of the formal model comply with the formal specification, thus leveraging the proof to a Coq one without C semantics.

DeepSpecDB already contained a formal model that complied with the abstract relation axiomatization [24]. However, this B+Tree model does not exactly mimic the behavior of the C code. For instance, entries of the C implementation contain a pointer to the child with greater keys. Then, the node contains a pointer to the first child. However, in this formal model, each entry contains a node which keys are lesser, and the node holds the pointer to the last child.

Moreover, the `First` and `Last` booleans used to speed up the functions are not present in each node, and not used in the functions. As a result, we decided to define another formal model, as close as possible to the C code. This new formal model will be used to specify the C functions with VST.

```
(* Btree Types *)
Inductive entry (X:Type): Type :=
    | keyval: key → V → X → entry X
    | keychild: key → node X → entry X
with node (X:Type): Type :=
    | btnode: option(node X)→ listentry X→ bool→ bool→ bool→ X→ node X
with listentry (X:Type): Type :=
    | nil: listentry X
    | cons: entry X → listentry X → listentry X.

Definition cursor (X:Type): Type := list (node X * index). (* ancestors and index *)
Definition relation (X:Type): Type := node X * X. (* root and address *)
```

Fig. 3. Coq Formal Model Types

Types **Fig.** 4 presents the Coq Types for B+Trees, cursors and relations. We can see that entries have a key, and either a record (of type `V`) or a child (of type `node X`). Nodes have three booleans, representing the `isLeaf`, `First` and `Last` of the C code. The ptr_0 of a node is represented with an option, as Leaf Nodes don't have any. These types are parametrized by a type `X`, that can be either `val` or `unit`. An explanation is given section 5.2.

A cursor is implemented in Coq as a list of nodes and indexes. This corresponds to the arrays found in the C code. The list starts at the root and its head is the current node and index. Its length indicates the cursor's depth. Finally, a relation is simply a root (node) and the address at which the representation is in the memory.

Functions Then, each C function must have an equivalent in the formal Coq model. For instance, **Fig.** 4 presents the Coq `moveToFirst` function. It takes as input the next node to go down to. If this node is a Leaf Node, then it returns the cursor with the new node, and the index 0 (pointing to the first record). Otherwise, it goes down ptr_0, and adds to the cursor the next node and the index `im` (representing -1).

5 VST Verification of B+Trees with Cursors

5.1 Using the Verified Software Toolchain

The Verified Software Toolchain uses Verifiable C to prove correctness of C programs. Verifiable C consists of a language and a program logic. The language of

```
(* takes a PARTIAL cursor, n next node (pointed to by the cursor)
   and goes down to the first key *)
Fixpoint moveToFirst {X:Type} (n:node X) (c:cursor X) (level:nat): cursor X :=
  match n with
    btnode ptr0 le isLeaf First Last x ⇒
    match isLeaf with
    | true ⇒ (n,ip 0)::c
    | false ⇒ match ptr0 with
               | None ⇒ c     (* not possible, isLeaf is false *)
               | Some n' ⇒ moveToFirst n' ((n,im)::c) (level+1)
               end
    end
  end.
```

Fig. 4. MoveToFirst in the formal model

Verifiable C is a subset of CompCert's Clight [1]. Clight was introduced as an intermediate language in CompCert, where all expressions are pure and side-effect free. The program logic of Verifiable C is a higher-order Separation Logic [26] (an extension of Hoare Logic [20]). VST includes many tools, proved theorems and Coq tactics to assist the user in writing a forward separation logic proof in Coq that relates Clight's semantics (as defined in CompCert) and a formal specification.

When proving the correctness of a C program, a VST user should first generate an equivalent Clight program. CompCert includes the `clightgen` tool to do so. Then, the user should write a specification in Coq for each C function. Finally, the user can prove the correctness of each function separately.

Verifiable C's Separation Logic. Hoare Logic has been extensively studied and used. The correctness of a program is represented by a Hoare triple $\{P\}\ c\ \{Q\}$, where P and Q are formulas (respectively called precondition and postcondition), and c is a program. Hoare Logic's simple rules allow to derive Hoare triples. For instance, the composition rule states that, if $\{P\}\ c_1\ \{Q\}$ and $\{Q\}\ c_2\ \{R\}$ hold, then $\{P\}\ c_1; c_2\ \{R\}$ holds.

Separation Logic is an extension of Hoare Logic. Correctness is still modeled with a triple $\{P\}\ c\ \{Q\}$. However, the formulas for preconditions and postconditions are augmented with a new operator $*$. Informally, $P_1 \ * \ P_2$ means that the heap (or memory) can be split into two disjoint parts, one where P_1 holds, and another where P_2 holds. This operator is convenient when dealing with multiples objects in the memory. For instance, if $\texttt{btnode\_rep}(n, p)$ means that the node n is represented in the memory at address p, then $\texttt{btnode\_rep}(n_1, p_1) * \texttt{btnode\_rep}(n_2, p_2)$ means that the two nodes are in the heap, at different addresses (in particular, $p_1 \neq p_2$). Without this operator, one would have to add many propositions of the form $p_1 \neq p_2$ when describing multiple objects in the heap, making the proof harder. Separation Logic also defines the *magic wand* operator $-\!\!*$. Informally, $P \;-\!\!*\; Q$ means that if the heap is extended with a disjoint part where P holds, then Q holds on the total heap. This is particularly useful to extract information from a separation construct, as seen in Section 5.3.

Function Specification in Verifiable C. In Verifiable C, a precondition or postcondition formula consists in three sets: `PROP`, `LOCAL` and `SEP`. `PROP` contains assertions of type `Prop` in Coq. `LOCAL` binds local variables to values. For instance, one could write `temp _a (Vint(Int.repr 0))` to state that local variable `a` is bound to 0. `SEP` contains spatial assertions in separation logic. Writing `P;Q` in `SEP` means that `P * Q` holds. **Fig.** 5 shows one of our specifications. The precondition contains several requirements, like `next_node c (get_root r) = Some n`, meaning that `n` is the node pointed to by the partial cursor. Comparing the `SEP` clauses of the precondition and postcondition allows to understand what happens in the memory. Here, the relation is unchanged, while the cursor is modified to represent the one returned by the Coq function `moveToFirst` (see **Fig.** 4).

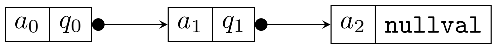
```
Definition moveToFirst_spec : ident * funspec :=
  DECLARE _moveToFirst
  WITH r:relation val, c:cursor val, pc:val, n:node val
  PRE[ _node OF tptr tbtnode, _cursor OF tptr tcursor, _level OF tint ]
    PROP(partial_cursor c r; next_node c (get_root r) = Some n)
    LOCAL(temp _cursor pc; temp _node (getval n);
          temp _level (Vint(Int.repr(Zlength c))))
    SEP(relation_rep r; cursor_rep c r pc)
  POST[ tvoid ]
    PROP()
    LOCAL()
    SEP(relation_rep r; cursor_rep (moveToFirst n c (length c)) r pc).
```

Fig. 5. Formal Specification of the `moveToFirst` function

5.2 Augmented Types

For each formal type, we need a representation predicate to specify how it is represented in the memory. For instance, in the VST verification of Binary-Search Trees [7], the predicate `tree_rep (t: tree val) (p: val) : mpred` is defined. Such predicates relate a formal structure to a statement on the content of the memory. However, this statement (of type `mpred`) holds more information than the formal model.

For instance, consider the following linked list in a C program:



This list is a representation of the formal list $[a_0, a_1, a_2]$. Each element contains a pointer to the following one. The pointers q_0, q_1 should not appear in the formal model but are needed by the representation predicate. One should write `listrep` such that this additional representation information is hidden from the user.

The usual way to do that in VST (or in any separation logic) consists in existentially quantifying over these values. For instance, one could write:

```
Fixpoint list_rep (l:list A) (p:val) : mpred :=
  match l with
    | [ ] ⇒ emp
    | ai:: l’ ⇒ EX pi:val, cell_rep ai pi p ∗ list_rep l’ pi
  end.
```

Where `cell_rep ai pi p` means that there exists a cell in the memory containing value `ai` and pointer `pi` at address `p`. `list_rep l p` describes that list `l` is represented in the memory, starting at address `p`. This is convenient for many data-structures. However, if an external data-structure holds pointers to the cells, we would want the values of such pointers to be the same as the ones quantified over. For instance, if some structure contains a pointer to the cell containing a_1, we need the pointer to be equal to q_0. But with the previous definition of `list_rep`, q_0 isn’t known outside of the quantifier’s scope.

We suggest using an augmented type for lists that holds both the formal model and the additional representation information. We first define the type `concrete_list` as `Definition concrete_list A : Type := list (A ∗ val)`. We then write an erasure function of type `concrete_list A → list A` (here the function `map fst`). We can change `list_rep` as follows:

```
Fixpoint list_rep (l:concrete_list A) (p:val) : mpred :=
  match l with
    | [ ] ⇒ emp
    | (ai,pi):: l’ ⇒ cell_rep ai pi p ∗ list_rep l’ pi
  end.
```

Finally, the formal structure of a pointer to a cell is an augmented cell (an element of the augmented list, of type `A * val`) and its representation can use the `val`, which is guaranteed to be the address of the cell in the memory.

In our B+Trees library, we have cursors containing pointers to subnodes of a B+Tree. We thus define a general B+Tree type, as seen on **Fig.** 4, parametrized with a type `X`. Finally, we define the following types:

```
Definition concrete_tree : Type := node val.
Definition abstract_tree : Type := node unit.
```

We then define a representation lemma on augmented trees and augmented cursors (in the appendix, **Fig.** 9). We also prove multiple lemmas about these representations. For instance, if a node is represented in the memory at some address p, then p is a valid pointer.

5.3 B+Trees Verification

We define properties of the B+Trees and cursors that are required to prove correctness. Such definitions include `partial_cursor c r`, meaning that `c` is a correct cursor for relation `r` that stops at an internal node. By correct, we mean that each node is the n^{th} child of the previous node, where n is the previous index.

We then prove multiple lemmas to help us reason with the memory representations of each structure. For instance, the function `currNode` returns a pointer to the last node of a cursor. In the verification proof, we need to access this pointer by proving that the current node is represented somewhere in the memory. However, the function precondition only states that the root node is represented in the memory. To solve this, we first need to prove that the current node is a subnode of the root. This is done by proving the following theorem:

```
forall X (c:cursor X) r,
complete_cursor_correct_rel c r → subnode (currNode c r) (get_root r)
```

We then need to prove that, if a node *root* is represented in the heap, and some other node n is a subnode of *root*, then n is also represented in the heap. This is true because the function `btnode_rep` calls itself recursively on each child (see **Fig.** 9). However, because `btnode_rep` is a separating clause, we can't simply use an implication. We need to rewrite it as a separation conjunction. This is done by proving the following theorem, **subnode_rep**:

```
forall n root, subnode n root →
btnode_rep root = btnode_rep n * (btnode_rep n -*btnode_rep root)
```

This separating conjunction means that some part of the heap contains the subnode `n`, and a disjoint part contains the rest of the B+Tree. Here, instead of defining another formal structure for an incomplete B+Tree, we use the magic wand operator. After proving many such theorems and writing every function specification (see **Fig.** 5), we need to prove the correctness of each function. These proofs consist in proving a Separation triple $\{P\}\ c\ \{Q\}$, where P and Q are the precondition and postcondition defined in the specification, and c is the Clight function. Each triple is proved by moving forward through c. For instance, if the first instruction of c is an assignment `a:=0`, using the `forward` tactic of VST will turn the proof goal $\{P\}$ `a=0;` c' $\{Q\}$ into $\{P'\}\ c'\ \{Q\}$, where P' is the strongest provable postcondition: P with the new `LOCAL` binding `temp _a (Vint(Int.repr 0))`. Occasionally, other goals have to be proven. Every time an array is accessed, we must prove that the index is in the right range. Every time a pointer is dereferenced, we must prove that it is valid. Even if VST is able to infer the next precondition to use for sequences of instructions, we must still provide loop invariants when going through a loop or a branching statement. When calling another function of the B+Trees library, we must prove that the precondition of this function holds. Finally, when we went through every statement of c, we are left with a goal $\{P\}$ `skip` $\{Q\}$, where Q is the function postcondition, and P is the new precondition obtained after moving through the program. Such goals amount to logical entailments ($P \vdash Q$) and are proved using separation logic rules implemented in VST.

5.4 Results

Every C function has been proved correct using VST with regards to its specification. There are still some admitted theorems that are used in these proofs. However, these are results about the formal model, and could all be proved without knowledge of VST or C semantics. The library, formal model and correctness proofs can all be found in the DeepSpecDB repository [2]. Overall, this represents more than 7 kloc of Coq and 1 kloc of C. In the appendix, Section 8.5 presents the VST proof of the function creating a new node.

6 Bugs of the original C implementation

Working on the verification of the B+Trees library has allowed us to find and fix some bugs. This demonstrates the benefits of formal verification: even though the library originally came with numerous tests, some bugs were only found when working with VST.

Wrong Array Access Originally, the library included a function, `findChildIndex`. Given a key and a sorted list of entries, this simple function would return the index at which the key should be inserted. It was used to go down the tree or insert a new key and record. Because that function could be called on internal nodes, it could return -1, if the key was strictly less than any key in the entry list. When going down the tree, it would mean that the next node to consider was the ptr_0 of the current node.

However, that function was also called on leaf nodes, that do not have a ptr_0. Then, in the insertion function, to check if the key was already inside the relation, we could see the following line for leaf nodes:

```
if (node->entries[targetIdx].key == key)
```

where `targetIdx` was the return value of `findChildIndex`. This means that the array could sometimes be accessed at index -1.

We fixed this issue by implementing another function `findRecordIndex` used on leaf nodes, which is formally proved to return a positive number.

Constructing Cursors for a New Key Another issue arose when building a new cursor for a key that wasn't already in the B+Tree. According to the formal specification, doing so should create a cursor that points just before the next key in the B+Tree. This means in particular that building such a cursor, then using the `GetRecord` function should return the record of the next key (if any).

However, in the original implementation, this wasn't enforced. Indeed, if the built cursor was at the end of a leaf node, the `GetRecord` function would access the last record of this node, instead of moving to the next leaf node. An example is given in the appendix, Section 8.2.

This was fixed by introducing equivalent positions for the cursor in a B+Tree. Informally, a cursor pointing at the end of a leaf node is equivalent to the one pointing at the beginning of the next leaf node. Previously, there was no difference between pointing at the last key of a node and after this key. This change requires the `MoveToNext` function to move to the next position twice if the cursor is pointing at the end of a leaf node. This isn't an issue for the complexity, as one of these moves is guaranteed to be constant-time.

Reducing complexity The original implementation of the `MoveToKey` function used to start from the root of the B+Tree, thus not exploiting the current position of the cursor. This was modified. Similarly, the insertion function did not split the nodes exactly in the middle if the fanout value was even. This does not affect the correctness of the B+Tree (which is still sorted and balanced) but can lead to a bigger depth on average, thus increasing the complexity of each function.

Structure Changes Working on the verification gave us the opportunity to clarify the entire structure of the implementation. We changed the types of B+Trees to remove superfluous fields. We factorized some functions. We clarified the notion of invalid cursor. Overall, more than 70% of the original library was rewritten.

Other changes for the verification Other changes were made to the original library to make verification possible. For instance, VST doesn't allow to verify an assignment where the type is a user-defined structure. We had to replace each entry copy (in the `splitnode` function for instance) with copies of each field, in order to prove the correctness.

Wrong function call The implementation of `moveToPrevious` used to call `moveToFirst` instead of `movetToLast`, meaning that the cursor would be moved several positions backward. Because this function wasn't tested, this simple bug was only found when doing the verification.

7 Conclusion and Future Work

We defined a Coq formal model for B+Trees with cursor, and proved using VST that the C implementation is correct with regards to this model. The verification work has allowed us to find and fix several bugs. Some properties of the formal model have yet to be proved. An example of such properties is that any complete cursor's length is equal to the depth of its tree. We are confident that these admitted theorems should be provable. All of these proofs only deal with the formal model, and could be done without using VST.

Like MassTree [23], DeepSpecDB uses the B+Trees library as a client to a Trie Library. A proof of this Trie library using VST is ongoing. The proof uses the abstract relation specification described in [24], and a proof that the formal model described in this work complies with it.

A feasible next step would be to implement and verify a concurrent library. We believe that both the implementation and the verification proof could be based on the work that has been presented in this paper.

References

1. CompCert Module Clight. http://compcert.inria.fr/doc/html/Clight.html.
2. DeepSpecDB. https://github.com/PrincetonUniversity/DeepSpecDB.
3. MemSQL. https://www.memsql.com/.
4. The CompCert Project. http://compcert.inria.fr/.
5. The Coq Proof Assistant. https://coq.inria.fr/.
6. The SQLite Database Engine. http://www.sqlite.org/.
7. Verified Software Toolchain. http://vst.cs.princeton.edu/.
8. VoltDB. https://www.voltdb.com/.
9. Oluwatosin Adewale. Implementing a High-Performance Key-Value Store using a Trie of B+Trees with Cursors. Master's thesis, Princeton University, 2018.
10. Andrew Appel. *Verifiable C: Applying the Verified Software Toolchain to C programs*, 2018.
11. Andrew W. Appel. Efficient Verified Red-Black Trees. https://www.cs.princeton.edu/ appel/papers/redblack.pdf.
12. Andrew W. Appel. Software Foundations Volume 3: Verified Functional Algorithms. https://softwarefoundations.cis.upenn.edu/.
13. Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, 2015.
14. Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.
15. Véronique Benzaken, Évelyne Contejean, Chantal Keller, and Eunice Martins. A Coq formalisation of SQL's execution engines. In *International Conference on Interactive Theorem Proving (ITP)*, Oxford, United Kingdom, July 2018.
16. Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 207–221, 2015.
17. Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
18. Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.
19. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 370–384, 2004.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
21. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54, 2006.
22. William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. *PACMPL*, 1(OOPSLA):87:1–87:28, 2017.
23. Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196, 2012.

24. Brian McSwiggen. The Theory and Verification of B+Tree Cursor Relation, 2018.
25. Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.
26. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.
27. Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 275–287, 2015.

8 Appendix

8.1 B+Tree insertion

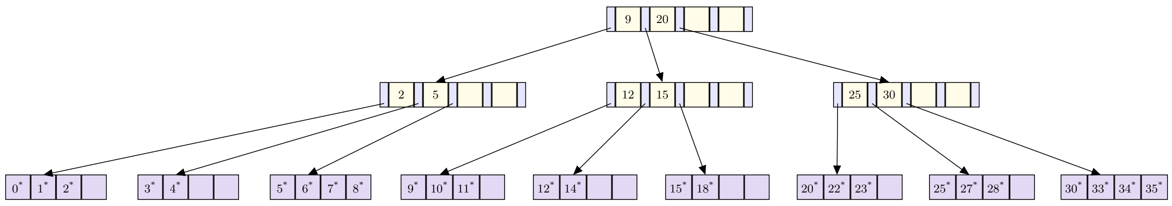


Fig. 6. The B+Tree of **Fig.** 1 after inserting a new record for key 4

8.2 Moving a cursor to a new key

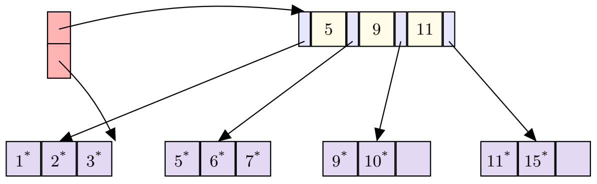


Fig. 7. Moving the cursor to key 4

In the B+Tree **Fig.** 7, moving the cursor to key 4 should make it point to the end of the first leaf node. Then, in the original implementation, accessing the record pointed to by the cursor would return the one associated with key 3. According to the abstract specification, the returned value should be the one associated with the next key (here, 5).

8.3 B+Trees with cursors library

The functions that can be used by a client of the library are:

- `Relation_T RL_NewRelation(void);` creates a new, empty relation.
- `Cursor_T RL_NewCursor(Relation_T relation);` creates a cursor for a given relation, pointing to the first key.
- `Bool RL_CursorIsValid(Cursor_T cursor);` returns true if the cursor is pointing to a key.
- `Key RL_GetKey(Cursor_T cursor);` returns the key pointed to by a cursor.
- `const void* RL_GetRecord(Cursor_T cursor);` returns the record pointed to by a cursor.
- `void RL_PutRecord(Cursor_T cursor, Key key, const void* record);` inserts a new record.
- `Bool RL_MoveToKey(Cursor_T cursor, Key key);` moves the cursor to a given key.
- `Bool RL_MoveToFirst(Cursor_T btCursor);` moves the cursor to the first key.
- `void RL_MoveToNext(Cursor_T btCursor);` moves the cursor to the next position.
- `void RL_MoveToPrevious(Cursor_T btCursor);` moves the cursor to the previous position.
- `Bool RL_IsEmpty(Cursor_T btCursor);` returns true if the B+Tree of the cursor is empty.
- `size_t RL_NumRecords(Cursor_T btCursor);` returns the number of keys in the B+Tree.

Fig. 8. B+Trees Library Functions

8.4 Representation of B+Trees in Separation Logic

```
Fixpoint entry_rep (e:entry val): mpred:=
  match e with
  | keychild _ n ⇒ btnode_rep n
  | keyval _ v x ⇒ value_rep v x
  end
with btnode_rep (n:node val):mpred :=
  match n with btnode ptr0 le b First Last pn ⇒
  EX ent_end:list(val ∗ (val + val)),
  malloc_token Tsh tbtnode pn ∗
  data_at Tsh tbtnode (Val.of_bool b,(
                       Val.of_bool First,(
                       Val.of_bool Last,(
                       Vint(Int.repr (Z.of_nat (numKeys n))),(
                       match ptr0 with
                       | None ⇒ nullval
                       | Some n' ⇒ getval n'
                       end,(
                       le_to_list le ++ ent_end)))))) pn ∗
  match ptr0 with
  | None ⇒ emp
  | Some n' ⇒ btnode_rep n'
  end ∗
  le_iter_sepcon le
  end
with le_iter_sepcon (le:listentry val):mpred :=
  match le with
  | nil ⇒ emp
  | cons e le' ⇒ entry_rep e ∗ le_iter_sepcon le'
  end.
```

Fig. 9. Representing a B+Tree node in the memory

8.5 A VST proof

```
static BtNode* createNewNode(Bool isLeaf, Bool First, Bool Last) {
     BtNode* newNode;
     newNode = (BtNode*) surely_malloc(sizeof (BtNode));
     if (newNode == NULL) {
       return NULL;
     }
     newNode->numKeys = 0;
     newNode->isLeaf = isLeaf;
     newNode->First = First;
     newNode->Last = Last;
     newNode->ptr0 = NULL;
     return newNode;
}
```

Fig. 10. The C code for creating a new node

```
Definition empty_node (b:bool) (F:bool) (L:bool) (p:val):node val := (btnode val) None (nil val) b F L p.

Definition createNewNode_spec : ident * funspec :=
  DECLARE _createNewNode
  WITH isLeaf:bool, First:bool, Last:bool
  PRE [ _isLeaf OF tint, _First OF tint, _Last OF tint ]
    PROP ()
    LOCAL (temp _isLeaf (Val.of_bool isLeaf);
         temp _First (Val.of_bool First);
         temp _Last (Val.of_bool Last))
    SEP ()
  POST [ tptr tbtnode ]
    EX p: val, PROP ()
    LOCAL (temp ret_temp p)
    SEP (btnode_rep (empty_node isLeaf First Last p)).
```

Fig. 11. Creating a new node in the formal model and specification of the C function

```
Lemma body_createNewNode: semax_body Vprog Gprog f_createNewNode createNewNode_spec.
Proof.
  start_function.
  forward_call (tbtnode).     (* t'1=malloc(sizeof tbtnode) *)
  - split. simpl. rep_omega.
    split; auto.
  - Intros vret.
    forward_if (PROP (vret≠ nullval)
    LOCAL (temp _newNode vret; temp _isLeaf (Val.of_bool isLeaf);
    temp _First (Val.of_bool First); temp _Last (Val.of_bool Last))
    SEP (malloc_token Tsh tbtnode vret ∗ data_at_ Tsh tbtnode vret)).
    + forward.                (* return null *)
    + forward. entailer!.
    + Intros.
      forward.                (* newNode→ numKeys = 0 *)
      unfold default_val. simpl.
      forward.                (* newnode→ isLeaf=isLeaf *)
      forward.                (* newnode→ First=First *)
      forward.                (* newnode→ Last=Last *)
      forward.                (* newnode→ ptr0=null *)
      forward.                (* return newnode *)
      Exists vret. entailer!.
      Exists (list_repeat Fanout (Vundef, (inl Vundef):(val+val))).
      simpl. cancel.
      apply derives_refl.
Qed.
```

Fig. 12. Proving the function correct with VST