PRINCETON UNIVERSITY

The B+-tree Index as a Verified Software Unit

by

Anastasiya Kravchuk-Kirilyuk

A Master's Thesis Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Advisor: Professor Andrew Appel

May 2020

Abstract

Ordered indexes are used in relational databases to allow for a faster retrieval of records. An ordered index can be implemented using a variety of data structures, and the query planner can later decide which implementation to use for optimal performance. This means the implementation details of an ordered index need not be exposed to the user. In this work, we define an API that any instance of an ordered index should implement. Then, we show that a particular instance of an ordered index, the B+-tree with cursors, satisfies this general interface. Finally, we present the B+-tree index instance as a Verified Software Unit, a VST-verified module with explicitly defined sets of imported and exported functions. That is, a B+-tree implementation verified for functional correctness to a B+-tree specification in previous work, is formally proved to be functionally correct w.r.t. the abstract specification of an ordered index. We verify our code using the Coq proof assistant, the Verified Software Toolchain (VST), and CompCert.

Acknowledgements

I would like to thank my advisor, Professor Andrew Appel, for his generous help and guidance over the past two years. I learned a lot from this project, but it pales in comparison with how much I learned from you as a person.

I would like to thank my reader, Lennart Beringer, for his enthusiastic involvement in my thesis project and for sharing his expertise.

I would like to thank Professor Benjamin Pierce and Professor Stephanie Weirich for introducing me to the world of verification in the first place.

Finally, I would like to thank my parents, Yelizaveta Makhatadze and Yaroslav Kravchuk-Kirilyuk, for their unconditional love and support, and for the many sacrifices they made for the sake of my future. Everything I have and everything I am now, I owe directly to you.

Contents

Chapter 1

Introduction

The various benefits of modular code composition extend not just to software implementation, but also to its verification. There are various situations where it is more suitable (and easier!) to compose verified modules together rather than perform verification from scratch. This allows us to reuse verified components and to build on already existing proofs. One scenario in which we can benefit from modular verification is an implementation of database operations using indexes for faster retrieval of records. For each type of index (ordered or unordered), there could exist many different implementations with various underlying data structures. Most of the time, the user need not concern themselves with the particular implementation of an index, since the database query planner will automatically choose the implementation with the best performance for a given query. Thus, when verifying database indexes and client code using those indexes, we would prefer to verify the code with respect to a general definition of an ordered index (as opposed to verifying several versions of the code with respect to particular instances of an ordered index). By introducing a separate layer of abstraction - an ordered index interface - we can ensure that any instance of an ordered index can be verified to a general ordered index specification, and that any client code verified with respect to the general specification will function as expected when using any instance of an ordered index. This layer of abstraction also ensures that no unnecessary implementation details are exposed to the client.

1.1 Contributions

We make the following contributions in this work:

- We define an API that serves as a generalization of an ordered index. Any implementation of an ordered database index can implement this API.
- We show that a particular instance of an ordered index, the B+-tree with Cursors, satisfies this general API by verifying the B+-tree functions with respect to the general ordered index specifications. This constitutes a formal proof that the B+-tree data structure, already verified to a specification of B+-trees, is therefore an instance of a general ordered index.
- We present the B+-tree index instance as a Verified Software Unit [1], a VST-verified module with explicitly defined sets of imported and exported functions. The exports of this module are the set of B+-tree interface functions which satisfy the functionality specified by our ordered index API, and are verified w.r.t. the general ordered index specifications. Thus, the exports of this module can be used as ordered index functions (with hidden implementations) by client modules.

This work is part of the DeepSpecDB project at Princeton University. All relevant code can be found in the DeepSpecDB Github repository [2].

Chapter 2

Background

2.1 Database Indexes

The relational model of databases has been the model of choice since the 1990s [3]. In a relational database, all records are stored in tables, which represent a **relation** on a set of data. Each table has a set of **attributes**, one per column, which describe the information that is stored in a given column. Each row in the table is a **tuple** representing a single data record. Any given tuple has one **component** per each attribute in the table. A **key** on a relation is a set of attributes that can be used to isolate a single record or a set of records within a relation. Depending on the type of key, different constraints apply to the values in the attributes of the key. For example, a **primary key** must be unique for every tuple in the relation, allowing each tuple to be uniquely identified using the primary key. The primary key also does not allow NULL values of attributes. On the other hand, a **secondary key** is a set of any attributes with potentially non-unique values. A secondary key can be used for purposes such as filtering or sorting.

Consider the following mini-example of a relation [3].

title	year	length	genre
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

In this example, we are storing information about movies in a relational database. The information we want to store includes the title of the movie, the year of release, the length of the movie in minutes, and the movie genre. Here, **title**, **year**, **length**, and

genre are attributes. An example of a tuple representing a single record is **(Star Wars, 1977, 124, sciFi)**. A primary key on this relation could include the attributes **title** and **year** (we assume that no two movies with the same title are released in the same year). A secondary key could include a single attribute **length**, and we could use this key for the purpose of finding the longest movie in the data set. Since **length** is a non-unique attribute, we might end up with several longest movies as a result.

When the user makes a query that involves sorting or filtering the tuples by their primary or secondary key, it is often useful to create an **index** on the data. The index is a temporary data structure which maps **keys** (the attributes of the primary or secondary key) to **values** (the tuples), and allows for a faster retrieval of records. Without an index, searching for a tuple in a relation takes linear time in the number of tuples. An index can be sorted or unsorted, which means different data structures can serve as database indexes with various benefits. Some examples of data structures that can be used as indexes are a hash table (constant-time lookup), a linked list (linear-time lookup), or a balanced search tree (B+-tree for short, log-time lookup).

2.2 B+-trees with Cursors

Ordered indexes are most useful for **range queries**. A range query returns a set of tuples which satisfy a **range condition** based on some key. For example, from our movie data set we could query all movies which have **length** between 90 and 120 minutes. Since there are likely many tuples that satisfy this range condition, we would not want to access these tuples from a data structure with linear lookup performance, like a linked list. A hash table would not work well either (even though it has constant-time lookup performance) because the tuples we are querying might not be hashed to nearby locations in the hash table. In this case, we would prefer to store the tuples in an ordered index, where the underlying data structure is sorted.

We will focus on the use of B+-trees with Cursors as ordered database indexes. In this data structure, the Cursor always keeps track of the most recent entry that was accessed in the B+-tree. We are also able to move the Cursor to the first entry in the B+-tree, and move it through the entries in the tree. The biggest performance benefit of this data structure is that accessing the next or previous element in the tree takes amortized constant time instead of logarithmic time. This is because the search for the next entry does not begin from the root every time, but is instead performed from the spot the cursor is pointing to. The B+-tree data structure can serve as an ordered index for integer keys. The main benefit of this particular implementation of ordered index is the quick retrieval of a set of consecutive tuples, sorted by key [4].

The code for B+-trees with Cursors has been implemented in C [4], and then verified for functional correctness in a prior work [5]. We build on this verification to show that this data structure satisfies our definition of an ordered database index. The relevant parts of the existing interface for B+-trees with Cursors are below. We will explain specific operations later, as we present their formal specifications.

FIGURE 2.1: B+-tree interface (relation.h)

```
typedef struct Relation* Relation_T;
typedef struct Cursor* Cursor_T;
typedef size_t Key;

/* Create a new relation */
Relation_T RL_NewRelation(void);

/* Delete the relation */
void RL_DeleteRelation(Relation_T relation, void (* freeRecord)(void *));

/* Create a cursor on the specified relation */
Cursor_T RL_NewCursor(Relation_T relation);

/* Free the cursor */
void RL_FreeCursor(Cursor_T cursor);

/* The cursor is invalid if it points after the biggest key */
Bool RL_CursorIsValid(Cursor_T cursor);

/* Get the key of the entry the cursor is currently pointing at */
Key RL_GetKey(Cursor_T cursor);

/* Put a key and its record into the relation */
void RL_PutRecord(Cursor_T cursor, Key key, const void* record);

/* Move the cursor to the position of key */
Bool RL_MoveToKey(Cursor_T cursor, Key key);

/* Get the record from the current cursor location */
const void* RL_GetRecord(Cursor_T cursor);

/* Delete key and its record */
Bool RL_DeleteRecord(Cursor_T cursor, Key key);

/* Move the cursor to the first record */
Bool RL_MoveToFirst(Cursor_T btCursor);

/* Go to the next record */
void RL_MoveToNext(Cursor_T btCursor);

/* Go to the previous record */
void RL_MoveToPrevious(Cursor_T btCursor);

/* Return True if the relation is empty. */
Bool RL_IsEmpty(Cursor_T btCursor);

/* Return the Number of Records in the Relation. */
size_t RL_NumRecords(Cursor_T btCursor);
```

2.3 Verification Tools

We verified our code using the Coq Proof Assistant [6], the Verified Software Toolchain (VST) [7], and CompCert [8]. VST is a verification tool which employs concurrent separation logic and allows us to verify the correctness of C programs directly inside the Coq proof assistant. Using the VST framework, we can write formal specifications for programs we want to verify (including preconditions, postconditions, and separation logic predicates). In order to verify that the body of a given C function adheres to its formal specification, VST needs access to the abstract syntax tree (AST) which represents the function. We utilize the front end of CompCert, a verified compiler for the C language, to transform a C function into a Clight AST, which VST can use for verification.

2.4 Verified Software Units

Our project employs Verified Software Unit calculus, a recent development in verifying modular programs using VST [1]. The VSU calculus allows for specifying and verifying modules in VST. VSU supports the principle of Abstract Data Types, in which a private data representation is accessed by several public interface functions (operations) and some private functions. Each module (or compilation unit) has its own Abstract Specification Interface which defines the functions exported by the module. Each module also specifies which functions are internal to the module (their definitions appear in the C file) and imported by the module (their definitions are imported from other files). Verifying a module using VSU calculus ensures that the internal function specifications are hidden from the client. The verified modules can also be composed in accordance with each module's specification interface. In essence, VSU calculus allows us to verify and compose compilation units in a modular way. In our case, the data representations are B+-trees and cursors, and the operations are those shown in Figure 2.1.

Chapter 3

Ordered Index Specification

In order to verify that a B+-tree data structure can serve as an ordered index in a relational database, we must first define a generalized notion of an ordered index. To serve as an ordered index, a data structure must allow an ordering to be imposed on its keys. Furthermore, the interface we define for an ordered index must meet the following desired criteria:

1. the interface must include essential ordered index functionality, such as lookup and insertion of tuples, and moving between the entries in the index
2. the interface must be general enough to accommodate various underlying data structures that can used to implement an ordered index
3. the interface must allow for use of any ordered index instance without disclosing unnecessary implementation details to the user (or requiring the user to understand the implementation)

3.1 Database Cursors

We define our generalized ordered index API using the classic notion of a **database cursor** and operations on a **cursor**. Database cursors are structures that allow for relation traversal [3]. This is similar to the concept of an iterator, which can traverse a relation by continuously returning the "next" tuple in the relation [3]. Unlike an iterator, a cursor does not have to return every tuple in the relation, but can instead range over select tuples. Besides traversal, database cursors can also perform modifying operations such as the lookup, insertion, and deletion of tuples [3]. Due to these similarities between the notion of a database cursor and the functionality we envisioned for the OrderedIndex interface, we heavily draw from the cursor concept in our implementation. In our implementation, a **cursor** contains two things: the underlying index **data structure** (B+-tree, Hashtable, Sorted Linked List, etc) and a **pointer structure** keeping track of the last accessed entry in the index. The pointer structure allows us to access tuples in the underlying data structure in sequential order, and offers potential performance benefits. The pointer structure can be a physical pointer, an array of pointers (this is the case for our B+-tree implementation), or even simply a number (for example, the number of a Linked List node in sorted order).

The use of cursors and iterators is essential when implementing database operators, such as **join**. When combining two relations with a **join**, one relation can be traversed through a cursor or iterator, while the other relation is accessed through individual lookups. One type of join is an **index join**, in which an index is created on one or both of the relations to speed up performance [3].

Note that we use the term "cursor" to refer to different things in different developments. A **B+-tree Cursor** (capitalized throughout the paper) is a B+-tree specific pointer structure, it is an array of pointers into the B+-tree keeping track of the last accessed node and all of its parent nodes. A **database cursor** (lowercase) is a general concept in databases, and it includes the underlying data structure as well as a pointer structure. For example, a particular implementation of a **database cursor** using B+-trees would consist of the B+-tree itself (the underlying data structure) and the B+-tree Cursor (the pointer structure). In this chapter, we highlight and use the general notion of **database cursors**.

3.2 Ordered Index Representation

We define the ordered index interface as a Coq record. In our definition, we include the following types and separation logic predicates, shown in Figure 3.1.

In our definition we include the type of the underlying data structure (**t**) in this particular instance of an ordered index, the type of the keys in this ordered index (**key**), and the type of values in this ordered index (**value**). In order to represent our ordered index functionality as operations on a cursor (as defined previously), we also include the type **cursor**. It might be unclear why we need representation for both **t** and **cursor**, since a **cursor** is implicitly a pointer into an index (a particular instance of the **t** type), so those operations that take a **cursor** argument don't need a separate argument of type **t**. However, our ordered index functionality includes creating an empty index as well as

FIGURE 3.1: Ordered Index Definition

```
Module OrderedIndex.
Record index :=
  {
    key: Type;
    key_repr: key -> val -> mpred;

    value : Type;
    value_repr: value -> val -> mpred;

    t: Type;
    t_repr: t -> val -> mpred;

    cursor : Type;
    cursor_repr: cursor -> val -> mpred;

    (* more components described later *)
  }
```

creating a new cursor on some index, and these functions take **t** as input rather than a **cursor**. The operations of creating a new index or a new cursor do not rely on a pointer into the underlying data structure (in fact, either no pointer exists yet, or a new pointer is being created), thus we choose not to have a meaningless **cursor** as input.

For each type (**key**, **value**, **t**, and **cursor**), we also define a separation logic representation (**value_repr**, **cursor_repr**, etc). These definitions are necessary in order to show the state of memory in the preconditions and the postconditions of our cursor operations. Separation logic predicates in the VST notation typically relate a mathematical value x of type τ to a data structure in memory at address p. Addresses are represented by CompCert's val type; therefore the representation predicate τ_repr would typically have type $\tau \to$ val $\to$ mpred [9]. We follow this style for all four of the representation predicates shown here.

3.3 Ordered Index Functionality

Our ordered index interface also includes the following functions, shown in Figure 3.3.

We define eight essential functionalities that any structure acting as an ordered database index must provide. There must be a way to create a new, empty index and also to create a new cursor on an index. Since a **relation** can be identified with its primary index, (**create_index**) also creates the empty relation. **create_cursor** must be a separate function, since any given index may have several active cursors on it at the same time. For example, there may be two different queries running on a single index, using two different cursors to step through the entries of the index. In order to find out how many tuples comprise the relation, we must be able to count how many entries are currently in an index (**cardinality**). We must also be able to search the relation for a single

Figure 3.2: Ordered Index Functionality

```
Module OrderedIndex.
Record index :=
  {
    key: Type;
    value : Type;
    t: Type;
    cursor : Type;

    (* continued from section 3.1 *)

    create_cursor: t -> cursor;
    create_index: t -> Prop;
    cardinality: cursor -> Z;
    go_to_key: cursor -> key -> cursor;
    move_to_next: cursor -> cursor;
    move_to_first: cursor -> cursor;
    get_record: cursor -> val;
    put_record: cursor -> key -> value -> val -> cursor -> Prop;

    (* other operations not shown here *)
  }
```

tuple, or a set of tuples, which match given values of the key attributes (**go_to_key**). **go_to_key** will move the cursor pointer to the first entry in the index that matches the search. In order to retrieve the tuple at the current cursor pointer position, we define a function **get_record**, which returns the location of the tuple as a memory pointer. We also define the functionality to move the cursor pointer along the underlying index data structure, (**move_to_next** and **move_to_first**. Finally, we must have the ability to insert new entries into the index, given a key-value pair (**put_record**).

In addition to the required functionality, we also define logical propositions that must hold true before each function is executed (**move_to_next_props**, **go_to_key_props**, etc). Since different implementations of an OrderedIndex might require different logical propositions in the precondition, we allow each implementation of an OrderedIndex to specify these separately.

3.4 Ordered Index Function Specifications

In order to prove (using VST) that a data structure satisfies our notion of an ordered index (meaning it has all of the required functionality to be used as an ordered database index), we must also define general function specifications for an ordered index. We will then be able to prove correctness of any instance of an ordered index (for example, the B+-tree index) with respect to these general function specifications. We show several examples of our ordered index specifications in Figure 3.3 below.

FIGURE 3.3: OrderedIndex funspecs

```
1   Definition create_cursor_spec
2     (oi: OrderedIndex.index): funspec :=
3     WITH r: t(oi), gv: globals, p: val
4     PRE [tptr t_type(oi)]
5       PROP(create_cursor_props(oi) r p)
6       PARAMS(p) GLOBALS(gv)
7       SEP(mem_mgr gv; t_repr(oi) r p)
8     POST [tptr cursor_type(oi)]
9       EX p':val,
10      PROP()
11      LOCAL(temp ret_temp p')
12      SEP(mem_mgr gv; oi.(t_repr) r p;
13           (oi.(t_repr) r p -* oi.(cursor_repr) (oi.(create_cursor) r) p')).
14
15  Definition move_to_next_spec
16    (oi: OrderedIndex.index): funspec :=
17    WITH p: val, cur: cursor(oi)
18    PRE [tptr cursor_type(oi)]
19      PROP(move_to_next_props(oi) cur)
20      PARAMS(p) GLOBALS()
21      SEP(cursor_repr(oi) cur p)
22    POST [tvoid]
23      PROP()
24      LOCAL()
25      SEP(cursor_repr(oi) (move_to_next(oi) cur) p).
```

In VST, each function specification (**funspec**) has a precondition **PRE** and a postcondition **POST**. The precondition must be met before the function body executes, and the postcondition must hold after execution. Each funspec also has a **WITH** clause which quantifies over Coq values that both the precondition and the postcondition have access to. The precondition specifies logical propositions (**PROP**) which must hold before execution, the input parameters to the function (**PARAMS**), the global variables present in the environment (**GLOBALS**), and separation logic predicates representing the memory state before execution (**SEP**). Similarly, the postcondition specifies logical propositions (**PROP**) and separation logic predicates (**SEP**) that hold after execution, and the return value of the function (the contents of the **ret_temp** variable in the **LOCAL** clause) [9].

For example, the funspec of the **create_cursor** function is defined as follows, starting at line 1. During the execution, we have access to **r** (the underlying index data structure), **p** (a pointer to **r** in memory), and **gv** (the global variables in the environment). These values are defined in the **WITH** clause on line 3. The type of input **create_cursor** takes is a pointer to the datastructure that serves as our index (line 4), and the output is a pointer to a cursor (line 8). Simply put, the function takes an index and creates a new database cursor on that index. Prior to program execution, there must be a representation in memory of data structure **r** at pointer **p** (line 7). After the program executes, there exists some pointer **p'**, which is the memory address for a newly created cursor on the data structure **r** (line 12). The global variables **gv** are unchanged, and the return value of the function is **p'** (line 11). The **mem_mgr** predicate is necessary in this specification since the function is allocating memory for a new cursor [10].

Note the use of the magic wand operator **-*** in the postcondition **SEP** clause (lines 12-13). In separation logic, the operators ***** (separating conjunction) and **-*** (separating implication) allow us to represent disjoint objects in memory. For example, **p -* q** means that given a disjoint structure **p** in memory, we have guarantees about a combination structure **q** in memory. We use the magic wand operator to clarify that the underlying data structure **t** and the **cursor** on this data structure are disjoint in memory. Using this design pattern allows us to have various cursors on the same data structure [11].

Similarly, the function specification of **move_to_next** requires that the existing cursor **cur** at memory location **p** is modified in place in a way that is defined by the functional model of **move_to_next**. This modification of a cursor in place is evident from the postcondition **SEP** clause on line 25. In this **SEP** clause, we reuse the existing cursor **cur** at pointer **p** instead of creating a new cursor. As a result, this funspec defines a void return type. This is an interesting point of difference between the funspec and the functional model we choose for **move_to_next**. The functional model must be able to represent the modified cursor in memory, which is why it has a **cursor** $\rightarrow$ **cursor** type. On the other hand, a C function need not return the same pointer it received as input (even if the structure at the pointer has been modified), as it is understood that the user has access to that pointer already.

As described above, our OrderedIndex interface is a Coq record comprised of type definitions and their corresponding separation logic predicates, function headers and their corresponding specifications, and a set of logical propositions that serve as preconditions to each function. Any instance of this OrderedIndex interface, when proven correct with respect to the function specifications in Coq, can be used as an ordered index on a relation. Next, we focus on the B+-tree instance of the OrderedIndex interface and its proof of correctness.

Chapter 4

B+-tree Index Specification

In a previous development, Barrière [5] proved the correctness of a C-language B+-tree implementation with Cursors, using VST. He proved that the implementation satisfied a functional correctness specification that was specific to B+-trees, although it was designed with the intention that it could satisfy some sort of general notion of an ordered index interface. Here, we take that intention and demonstrate it formally and concretely. We build upon Barrière's work to prove functional correctness of the B+-tree functions with respect to the more general OrderedIndex specification. Effectively, we show that the B+-tree with Cursors is a data structure that can be used as an ordered index in a database.

We hoped that it would be possible to re-use most of Barrière's proofs through **function specification subsumption** in VST [12]. We use subsumption when a single C function has two different specifications: a tighter specification and a looser specification. If we have a proof that the function adheres to the tighter specification, and we also show that the looser specification **subsumes** the tighter specification, then we can prove that the function also adheres to the looser specification. Thus, if we could show that our general OrderedIndex funspecs subsume the tight B+-tree funspecs, we could show that the B+-tree functions adhere to the general OrderedIndex specification.

We were able to build on most of Barrière's proofs through the use of function specification subsumption. This means we only had to show that the OrderedIndex specification of each function was weaker than the existing B+-tree specification, and thus subsumed the tighter specification. However, some of Barrière's function definitions posed a challenge. One issue we encountered was the lack of some B+-tree function body proofs. Some functions that comprised the B+-tree interface were not verified, but the helpers used inside those functions were verified. For these functions, we wrote tight B+-tree funspecs, completed the proofs w.r.t. the B+-tree funspecs, and then used subsumption to show that the OrderedIndex specs subsumed the B+-tree specs. Another, more difficult issue presented itself when two of the B+-tree functional models turned out to be insufficiently general. These functional models required too many arguments that were specific to the implementation, and we could not implement our generalized OrderedIndex interface using these models. In these two cases, we elected to represent the corresponding ordered index functionality as relations instead of functions (**create_index** and **put_record**).

Looking at the existing B+-tree code, we made the following connections between the B+-tree functions and the OrderedIndex functions, shown in Figure 4.1. In this chapter, we *italicize* the names of all B+-tree specific definitions, and format the names of all OrderedIndex definitions in **bold**. Note that **go_to_key** matches most closely with a helper function that has been verified in the B+-tree code, and does not have a B+-tree interface function equivalent. The third column describes the approach we used to adapt the existing B+-tree functions and body proofs to fit our B+-tree ordered index implementation. The three approaches are sorted in this table from the least involved (using the existing funspec and body proof) to most involved (representing functionality as a relation instead of using the existing, overly specific functional model).

FIGURE 4.1: Functions with corresponding functionality

B+-tree Code	OrderedIndex Interface	Approach
RL_NewCursor	**create_cursor**	existing spec
goToKey	**go_to_key**	existing spec
RL_MoveToNext	**move_to_next**	existing spec
RL_GetRecord	**get_record**	existing spec
RL_NumRecords	**cardinality**	fresh spec and proof
RL_MoveToFirst	**move_to_first**	fresh spec and proof
RL_NewRelation	**create_index**	relation representation
RL_PutRecord	**put_record**	relation representation

4.1 The B+-tree Index Implementation

We implement the B+-tree Index (**btree_index**) as a Coq record, which is an instance of the OrderedIndex record. In our definition, we include the following types and separation logic predicates (Figure 4.2):

Figure 4.2: B+-tree Index

```
Import OrderedIndex.
Definition btree_index : index :=
  {| key := btrees.key;
     key_repr := emp;

     value := btrees.V;
     value_repr := value_rep;

     t := relation val;
     t_repr := fun m p => !!(snd m = p) && relation_rep m;

     cursor := (bt_relation * bt_cursor)%type;
     cursor_repr := fun '(m, c) p => relation_rep m * cursor_rep c m p;

    (* more components described later *)
  }
```

The definitions of **key** and **value** types are simply the key and value types from the B+-tree development (*btrees.key* and *btrees.V*). The separation logic predicate for the memory representation of a **value** also directly corresponds to the B+-tree predicate *value_rep*. The separation logic predicate for the memory representation of a **key** is empty since B+-tree keys are integers (a primitive data type).

The index type **t** is a B+-tree, which we initialize as a tree with pointer values. Here, *relation* is a dependent type representing a B+-tree, and the value (*val*) corresponding to each key is a pointer to a single tuple. The separation logic predicate for the memory representation of a B+-tree (**t_repr**) is also already defined in the B+-tree development (*relation_rep*), but we include an additional assertion to ensure the structure is stored at a specific pointer **p** in memory.

The **cursor** type is a product type consisting of a B+-tree (*bt_relation*) and a B+-tree cursor (*bt_cursor*). The representation of a **cursor** in memory is a separating conjunction of the existing B+-tree representation predicate and the B+-tree cursor representation predicate. Once again, it is important to review the conceptual difference between **cursor** and *bt_cursor*: the latter is simply an array of pointers to some entry in the B+-tree, whereas the former is a tuple that contains both the B+-tree data structure and the array of pointers.

4.2 Functions with Matching B+-tree Specs

In Figure 4.3 are definitions of **btree_index** functions which we were able to seamlessly adapt using existing B+-tree funspecs. There were a total of four **btree_index** functions which had corresponding B+-tree functions with matching inputs, outputs, specs, and directly usable body proofs. This subset of functionality was the easiest to implement

Figure 4.3: Directly Usable Functions

```
Import OrderedIndex.
Record btree_index : index :=
  {
    (* continued from section 4.1 *)

    create_cursor := fun m => (m, (first_cursor (get_root m)));
    go_to_key := fun '(m, c) k => (m, goToKey c m k);
    move_to_next := fun '(m, c) => (m, (RL_MoveToNext c m));
    get_record := fun '(m, c) => RL_GetRecord c m;

    (* more components described later *)
  }
```

Figure 4.4: Funspec Subsumption Example

```
Definition moveToNext_funspec :=
  WITH c:cursor val, pc:val, r:relation val
  PRE[ tptr tcursor ]
    PROP(complete_cursor c r; correct_depth r;
         root_wf(get_root r); root_integrity (get_root r))
    PARAMS(pc) GLOBALS()
    SEP(relation_rep r; cursor_rep c r pc)
  POST[ tvoid ]
    PROP()
    LOCAL()
    SEP(relation_rep r; cursor_rep (RL_MoveToNext c r) r pc).

Definition move_to_next_spec
  (oi: OrderedIndex.index): funspec :=
  WITH p: val, cur: cursor(oi)
  PRE [ tptr cursor_type(oi)]
    PROP(move_to_next_props(oi) cur)
    PARAMS(p) GLOBALS()
    SEP(cursor_repr(oi) cur p)
  POST [tvoid]
    PROP()
    LOCAL()
    SEP(cursor_repr(oi) (move_to_next(oi) cur) p).
```

since we were able to simply show function specification subsumption of the existing B+-tree funspecs by our more general OrderedIndex funspecs.

In order to define this subset of **btree_index** functions, we were able to reuse the existing B+-tree functional model and the corresponding C functions without making any changes to the previously verified code. Next, we show an example of a B+-tree funspec which is subsumed by an OrderedIndex funspec.

4.2.1 Function Specification Subsumption

Consider the following two specifications for the same function, **move_to_next**, in Figure 4.4. The former is the tighter B+-tree function specification from the existing B+-tree code (*moveToNext_funspec*), whereas the latter is a general specification from the OrderedIndex interface (**move_to_next_spec**).

FIGURE 4.5: B+-tree Index Props

```
Import OrderedIndex.
Record btree_index : index :=
  {
    (* continued *)

    move_to_next_props := fun '(m, c) =>
        complete_cursor c m /\ correct_depth m /\
        root_wf (get_root m) /\ root_integrity (get_root m);

    go_to_key_props := fun '(m, c) =>
        complete_cursor c m /\ correct_depth m /\
        root_integrity (get_root m) /\ root_wf (get_root m);

    (* more components described later *)
  }
```

Note that is it easy to show how the latter subsumes the former given our definitions of **cursor**, **cursor_type**, **cursor_repr**, and **move_to_next** in the B+-tree Index record (which is an instance of the OrderedIndex interface). The two specifications have corresponding inputs and outputs, identical SEP clauses before and after function execution (since **cursor_repr** in **btree_index** is defined as a conjunction of *relation_rep* and *cursor_rep* predicates), and the cursor after execution is modified by the function **move_to_next** (which is defined in terms of *RL_MoveToNext* in the **btree_index** definition). One point of concern might be the logical propositions that must hold before function execution (the PROP clause). However, in our definition of **btree_index** we include a set of logical propositions for each function that requires it, e.g. **move_to_next_props** (Figure 4.5).

Thus, we ensure that all the required logical propositions are available at the beginning of function execution. With all these pieces in place, we can prove that the **btree_index** specification subsumes the existing B+-tree specification by doing a **funspec_sub** proof in VST [12]. For example, consider this lemma which uses the two funspecs above.

```
Lemma sub_move_to_previous: funspec_sub moveToPrevious_funspec
  (move_to_previous_spec btree_index).
```

Proving the above lemma shows that **move_to_previous_spec** (as it is defined for the **btree_index**) subsumes *moveToPrevious_funspec*. Our funspec subsumption proofs for each of the **btree_index** functions are available in the code base [2]. Together, these proofs show that B+-trees with Cursors satisfy our general definition of an OrderedIndex.

Figure 4.6: Functions with Fresh Specs

```
Import OrderedIndex.
Record btree_index : index :=
  {
    (* continued from section 4.2 *)

    cardinality := fun '(m, c) => get_numrec m;
    move_to_first := fun '(m, c) =>
        let (n, p) := m in (m, moveToFirst n empty_cursor 0);
  }
```

4.3 Functions with Fresh Specs

There were a few B+-tree functions and body proofs which we were not able to use as-is to instantiate our B+-tree index. The corresponding **btree_index** functions are **cardinality** and **move_to_first**. Their definitions in the **btree_index** instance are in Figure 4.6.

Even though we were once again able to use the existing B+-tree functional model for these definitions, the definitions are less elegant. Note that **moveToFirst** (which comes from the B+-tree functional model) requires a lot of very specific parameters, including the length of a B+-tree Cursor (the last parameter) and the root node of the B+-tree (n). This is because **moveToFirst** represents a helper C function, rather than a B+-tree interface function. In fact, the B+-tree interface function *RL_MoveToFirst* did not have a corresponding body proof in the existing development, but the helper function did. We were able to write a fresh spec for *RL_MoveToFirst* and verify this function using the existing body proof of its helper function. The B+-tree function *RL_NumRecords*, corresponding to **cardinality** in **btree_index**, did not have a funspec or a body proof either; we had to add both to the development as well. Finally, after we supplemented the missing B+-tree specs and body proofs, we could once again use function specification subsumption to show that the OrderedIndex specs subsume the B+-tree specs.

4.4 Functions Represented as Relations

The last two **btree_index** functions, **create_index** and **put_record**, proved most difficult to define due to incompatible parameter lists in the B+-tree functional model. For example, the Coq function *empty_relation*, which is used in the existing funspec for the C function *RL_NewRelation*, takes two pointers as arguments: a pointer to the B+-tree, and a pointer to the root node of the B+-tree. On the other hand, the OrderedIndex function **create_index** does not take in any pointers as arguments, since it is unnecessary to know the exact location of the B+-tree until it is created. If we did modify **create_index** to take in two pointers, the definition might not be general enough to accommodate any other type of index besides the B+-tree. Because **create_index** is not deterministic, it can choose to create the new empty index at any address that malloc() might return, it is not naturally modeled as a function. Thus, we modeled *empty_relation* as a relation (*empty_relation_rel*), and then used it to instantiate **create_index** in **btree_index**.

```
Definition empty_relation_rel newr: Prop :=
  exists pr pn, newr = empty_relation pr pn.
```

Here, *newr* is a B+-tree that satisfies the property *empty_relation_rel* if there exist two pointers such that *empty_relation* with those two pointers as inputs returns *newr*.

Similarly, the Coq function *RL_PutRecord* (which is the functional model for the C function *RL_PutRecord*) takes in seven different inputs.

```
Definition RL_PutRecord (c: cursor val) (r: relation val) (key: key)
    (record: V) (recordptr: val) (newx: list val) (d: val)
    : (cursor val * relation val) :=  (* omitted *).
```

Even though the first five inputs can be generalized to other index instances besides the B+-tree, the last two inputs are specific to the B+-tree implementation. It would be unreasonable to have a **put_record** OrderedIndex function with inputs that are instance-specific. Thus, we chose to represent *RL_PutRecord* as a relation as well. This relation is then used to instantiate **put_record** in **btree_index**.

```
Definition RL_PutRecord_rel (c:cursor val) (r:relation val) (key:key)
  (record:V) (recordptr:val) (newr: relation val) (newc: cursor val) : Prop :=
exists newx, (newc, newr) = RL_PutRecord c r key record recordptr newx nullval.
```

After we formulated the necessary relations and rewrote parts of the B+-tree code to support the modified functional model, we once again used funspec subsumption to show that the OrderedIndex specs for **create_index** and **put_record** subsumed the B+-tree specs. This completed the **btree_index** implementation.

Chapter 5

The B-tree Index as a VSU

As the final contribution of our work, we present the B+-tree Index as a Verified Software Unit [1]. The VSU calculus is a recently proposed approach to VST verification, using which different C compilation units can be modularly verified and composed together. This approach is highly beneficial when verifying database operations, considering the various implementations of database indexes that must be verified separately. Using VSU calculus, we can verify any instance of an index as a VST module such that only the specifications of the API-exported functions are exposed to the client, and any internal or imported specifications are hidden from the client. The exported specifications also serve as a guarantee to the client - when a funspec precondition is met, the postcondition specifies what should be expected after the function executes [1].

In this work, we use VSU calculus to wrap up the B+-tree instance of an OrderedIndex as a VST-verified module with explicitly stated imports, exports, and internal functions. Since the B+-tree C file consists of over 40 internal functions, this contribution is a new, larger example of using VSU calculus for modular verification, when compared to the previously verified examples [1]. Beyond the fact that any instance of an index can be verified as a VSU, we can further build upon our contributions to modularly verify database operations, as well as the client code that uses those operations.

We present the B+-tree Index as a VSU in two steps, mirroring the verification of the B+-tree Index with respect to the OrderedIndex specification. First, we wrap up the B+-trees with Cursors data structure as a module. Then, we make use of function specification subsumption to show that the chosen exportable API of the B+-tree data structure also satisfies the more general OrderedIndex specification. The latter module is the B+-tree Index presented as a Verified Software Unit.

Figure 5.1: B+-tree ASI

```
Section BtreeASI.

  Definition cardinality_funspec := RL_NumRecords_spec.
  Definition create_cursor_funspec := RL_NewCursor_spec.
  Definition create_index_funspec := RL_NewRelation_spec.
  Definition move_to_next_funspec := RL_MoveToNext_spec.
  Definition go_to_key_funspec := goToKey_spec.
  Definition move_to_first_funspec := RL_MoveToFirst_spec.
  Definition get_record_funspec := RL_GetRecord_spec.
  Definition put_record_funspec := RL_PutRecord_spec.

  Definition BtreeASI:funspecs :=
      [ cardinality_funspec; create_cursor_funspec;
        create_index_funspec; move_to_next_funspec;
        go_to_key_funspec; move_to_first_funspec;
        get_record_funspec; put_record_funspec ].

  Definition Btree_exportedFunIDs :list ident := map fst BtreeASI.

End BtreeASI.
```

Figure 5.2: B+-tree Component and VSU

```
Definition BtreeComponent: @Component NullExtension.Espec BtreeVprog _
      nil imported_specs prog BtreeASI internal_specs.
```

```
Definition BtreeVSU: @VSU NullExtension.Espec BtreeVprog _
      nil imported_specs prog BtreeASI.
```

5.1 The B+-tree Module

In Figure 5.1, we show the Abstract Specification Interface (ASI) of the B+-tree with Cursors, which specifies the functions that can be exported by this module.

Each of the funspecs in this ASI is specific to the B+-tree datastructure, as opposed to the general OrderedIndex funspecs. This ASI is the set of funspecs that can be **exported** by the B+-tree module. Furthermore, we define the set of funspecs internal to this module, **internal_specs**. Internal functions are the functions that have definitions in the given compilation unit, the B+-tree C file. As previously mentioned, there were over 40 internal functions in this module, a number vastly larger than in the previous examples showcasing the use of VSU calculus [1]. Finally, we define the set of funspecs imported by this module, **imported_specs**. The imported functions are the functions that are called from this module, but have definitions outside of this module. The imported functions in the B+-tree module were **free**, **malloc**, and **exit**.

Using the definitions of **BtreeASI**, **imported_specs**, and **internal_specs**, we defined and verified the **BtreeComponent** (5.2). Essentially, the BtreeComponent is a proof that a compilation unit satisfies given function specifications and adheres to the defined sets of exported, imported, and internal functions.

FIGURE 5.3: B+-tree Index ASI

```
Import OrderedIndex.
Section BtreeIndexASI.
  Variable BtreeIndexPreds: index.

  Definition cardinality_funspec :=
    (_RL_NumRecords, cardinality_spec btree_index).
  Definition create_cursor_funspec :=
    (_RL_NewCursor, create_cursor_spec btree_index).
  Definition create_index_funspec :=
    (_RL_NewRelation, create_index_spec btree_index).
  Definition move_to_next_funspec :=
    (_RL_MoveToNext, move_to_next_spec btree_index).
  Definition move_to_previous_funspec :=
    (_RL_MoveToPrevious, move_to_previous_spec btree_index).
  Definition go_to_key_funspec :=
    (_goToKey, go_to_key_spec btree_index).
  Definition move_to_first_funspec :=
    (_RL_MoveToFirst, move_to_first_spec btree_index).
  Definition get_record_funspec :=
    (_RL_GetRecord, get_record_spec btree_index).
  Definition put_record_funspec :=
    (_RL_PutRecord, put_record_spec btree_index).

  Definition BtreeASI:funspecs :=
      [ cardinality_funspec; create_cursor_funspec;
        create_index_funspec; move_to_next_funspec;
        move_to_previous_funspec; go_to_key_funspec;
        move_to_first_funspec; get_record_funspec;
        put_record_funspec ].

  Definition BtreeIndex_exportedFunIDs :list ident := map fst BtreeIndexASI.

End BtreeIndexASI.
```

The proof of BtreeComponent required us to verify correctness of all **internal** functions w.r.t. the specs in **internal_specs**. Thus, this proof had 40+ subgoals. For each subgoal, we had to provide a proof that a given B+-tree function adheres to the provided B+-tree spec (mostly Barrière's proofs, plus the ones we wrote for *RL_MoveToFirst* and *RL_NumRecords*). Some of the B+-tree internal functions that had functionality beyond our B+-tree Index capabilities (e.g., *RL_DeleteRelation*) or used for debugging (e.g., *RL_PrintTree*) did not have existing B+-tree specs and body proofs. We had to use placeholder specs and admitted body proofs for those functions. It is important to note, however, that all of the functionality that was exported by the BtreeASI was completely verified with respect to the proper specifications. Finally, the **BtreeVSU** proof (5.2) is a one-line proof that finalizes the process of wrapping up the B+-tree Module as a VSU by applying BtreeComponent.

5.2 The B+-tree Index Module

In Figure 5.3, we show the Abstract Specification Interface (ASI) of the B+-tree Index, which specifies the functions that can be exported by this module.

Figure 5.4: B+-tree Index Component and VSU

```
Definition BtreeIndexComponent: @Component NullExtension.Espec BtreeVprog _
      nil imported_specs prog BtreeIndexASI internal_specs.
```

```
Definition BtreeIndexVSU: @VSU NullExtension.Espec BtreeVprog _
      nil imported_specs prog BtreeIndexASI.
```

Note that each function specification definition connects the name of a B+-tree function (for example, *_RL_NumRecords*) to an ordered index specification from the **btree_index** instance. This interface is thus verified with respect to the OrderedIndex specification. Since the only difference between BtreeASI and BtreeIndexASI are the specs of the exported functions (the C file used for this compilation unit is still the same), the sets of funspecs **internal_specs** and **imported_specs** remained the same as for the proofs in the previous section.

Using the definitions of **BtreeIndexASI**, **imported_specs**, and **internal_specs**, we defined and verified the BtreeIndexComponent (5.4).

We were able to prove **BtreeIndexComponent** and **BtreeIndexVSU** through VSU subsumption lemmas. Essentially, we showed that the B+-tree Index Module (BtreeIndexVSU) subsumes the B+-tree Module (BtreeVSU). For every function **f** exported by BtreeASI, we needed to prove that the BtreeIndexASI specification for **f** subsumed the BtreeASI specification for **f**. We were able to easily accomplish this using the funspec subsumption proofs we discussed in the previous chapter.

The encapsulation of a module using VSU calculus ensures that a certain level of modularity and abstraction is maintained throughout the code. It was entirely possible that while verifying the B+-tree Index VSU we could have run into many issues regarding the abstract layer (OrderedIndex) not being abstract enough, and the B+-tree implementation not being modular enough. However, the only minor change made to the code after we began verification with VSU calculus is the slight change to the **Gprog** that the B+-tree function body proofs used. The Gprog is just a list of funspecs w.r.t. which we verify the C functions in a compilation unit [9]. Originally, the B+-tree Gprog included both the internal and imported functions in one list, but we defined two separate lists in order to be able to use them seamlessly in the building of the Components. The fact that we had to make very minor changes shows that the B+-tree code and proof base was already sufficiently modular in nature, and our OrderedIndex interface was sufficiently abstract to begin with.

Chapter 6

Future Work

This approach to modular verification can be used to verify any suitable data structure as a database index with necessary properties. For example, another useful data structure in our development is a trie of B+-trees, which can serve as an ordered index on String keys. We could implement a **trie_index** instance of the OrderedIndex interface and wrap up the trie functions in a module using VSU calculus. This means the client code would be able to use the functionality of the trie ordered index without being exposed to the implementation details of the index. In fact, even different implementations of the same data structure (potentially with different performance benefits) could be modularly verified and selected by the query planner as deemed fit. For example, the user need not know which implementation of a **btree_index** they are using, since the most efficient version will be selected by the query planner. The same approach could be used for unordered indexes as well.

An extension of this project could address the OrderedIndex interface functionality. We already allow three traversal operations in our interface: **move_to_first**, **move_to_next**, and **go_to_key**. Some potential traversal operations we could implement further are **move_to_previous** and **move_to_last**. This functionality would allow the OrderedIndex to be traversed not just in the forward direction, but also in the backward direction. This functionality could be useful in an ordered index when searching for all tuples with keys below a certain value, or returning the last N tuples in the relation (sorted by key). Note that this functionality would not be as useful for an unordered index because an unordered index makes no guarantees about the order in which tuples are traversed. Some other desired functionality could be the deletion of a tuple, or an entire index. We originally planned to include this functionality in the OrderedIndex interface, but the lack of some B+-tree body proofs and C functions prevented us from doing so.

Another planned area of exploration is the verification of database operations, which use ordered and unordered indexes in their implementation. If we verify several data structures according to our OrderedIndex interface, we can abstract away the implementation of an OrderedIndex, and verify correctness of any database operation regardless of the OrderedIndex instance the query planner chooses to use. We discuss some database operations of interest below.

One of the simplest operations that can be performed on a relation is a **sequential scan**. This operation traverses through all tuples in the relation in sequential order [3]. If combined with a filter operation, the sequential scan may return only tuples that meet a certain filtering condition. Another similar operation is the **index scan**, during which an index on the data is created first and then a scan is performed [3]. The type of index created depends on the purposes of the scan. An index scan may result in better performance than a sequential scan, especially for queries which are concerned with an ordering on the tuples. Take the example of finding N longest movies in our movie database. Surely, creating a B+-tree index on the data and returning the N top entries in sorted order is more efficient than searching through the entire relation with a sequential scan. However, sequential scan is not always as inefficient as it sounds. The query planner might choose a sequential scan over an index scan if the set of data is small enough so that creating an index on it unnecessarily slows down performance, or if the output of the query is nearly the entire relation as opposed to a few matching tuples. In these scenarios, traversing all tuples in the relation without creating an index on it might be the more efficient choice.

Another operation of interest is a **join** on two relations. When the relations are joined, tuples with shared values of the key attributes are combined into longer tuples with attributes from both relations [3]. For example, if one relation in our movie database example has attributes (**title**, **year**, **length**, **genre**) and another relation has attributes (**title**, **year**, **director**), we can join tuples in the two relations based on the common attributes (**title**, **year**), resulting in tuples with attributes from both relations: (**title**, **year**, **length**, **genre**, **director**). If an index exists on either or both relations, it can speed up the join process. Sometimes, an index will be deliberately created on one of the relations to improve performance. Consider a scenario in which one of the relations has 100 entries, another has 1,000,000 entries, and we don't expect much overlap in the values of the common attributes. This means that the expected size of the joined relation is quite small (closer to 100 entries). The query planner might elect to create a hash table index on the larger relation in order to efficiently access the few tuples that match. On the other hand, if we are expecting a lot of of overlap in the values of the common attributes, we might create a sorted B+-tree index on the larger relation. This would allow us to iterate through tuples in the smaller relation, and easily access the subset of all matching tuples in the larger relation due to the ordering on the keys (instead of searching for each matching tuple individually).

Chapter 7

Conclusion

In this paper, we described the OrderedIndex API that we used to introduce a layer of abstraction between instances of an ordered index and the client code. We then used an existing implementation [4] and proof of functional correctness [5] of a B+-tree with Cursors to show that this data structure satisfies the OrderedIndex API we defined. In order to accomplish this, we proved function specification subsumption of existing B+-tree specifications by the more general OrderedIndex specifications, wrote specifications and body proofs for several B+-tree functions that were not previously verified, and made changes to the B+-tree functional model where necessary. Lastly, we defined an Abstract Specification Interface (ASI) for both the B+-tree and the B+-tree Index modules, and verified these components with the help of VSU calculus [1]. This final step completed our presentation of the B+-tree Index instance as a Verified Software Unit. The code and proofs for this development are available in our GitHub repository [2].

Bibliography

[1] Lennart Beringer. Verified Software Units. Unpublished Manuscript, 2020.

[2] DeepSpecDB Github Repository. `https://github.com/PrincetonUniversity/DeepSpecDB`.

[3] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2014.

[4] Oluwatosin Adewale. Implementing a High-Performance Key-Value Store using a Trie of B+Trees with Cursors. Master's thesis, Princeton University, 2018.

[5] Aurèle Barrière. VST Verification of B+Trees with Cursors. Technical report, Princeton University, 2018.

[6] The Coq Proof Assistant. `https://coq.inria.fr/`.

[7] Verified Software Toolchain. `http://vst.cs.princeton.edu/`.

[8] The CompCert Project. `http://compcert.inria.fr/`.

[9] Andrew Appel. *Verifiable C: Applying the Verified Software Toolchain to C programs*. 2020.

[10] Andrew W. Appel and David A. Naumann. Verified sequential malloc/free. In *ACM SIGPLAN International Symposium on Memory Management (ISMM 2020)*, 2020. To appear.

[11] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design Patterns in Separation Logic. *Proceedings of the 4th international workshop on Types in language design and implementation - TLDI 09*, 2008. doi: 10.1145/1481861.1481874.

[12] Lennart Beringer and Andrew W. Appel. Abstraction and Subsumption in Modular Verification of C Programs. *Lecture Notes in Computer Science Formal Methods – The Next 30 Years*, page 573–590, 2019. doi: 10.1007/978-3-030-30942-8_34.