Let's Build a Simple Database

Writing a sqlite clone from scratch in C

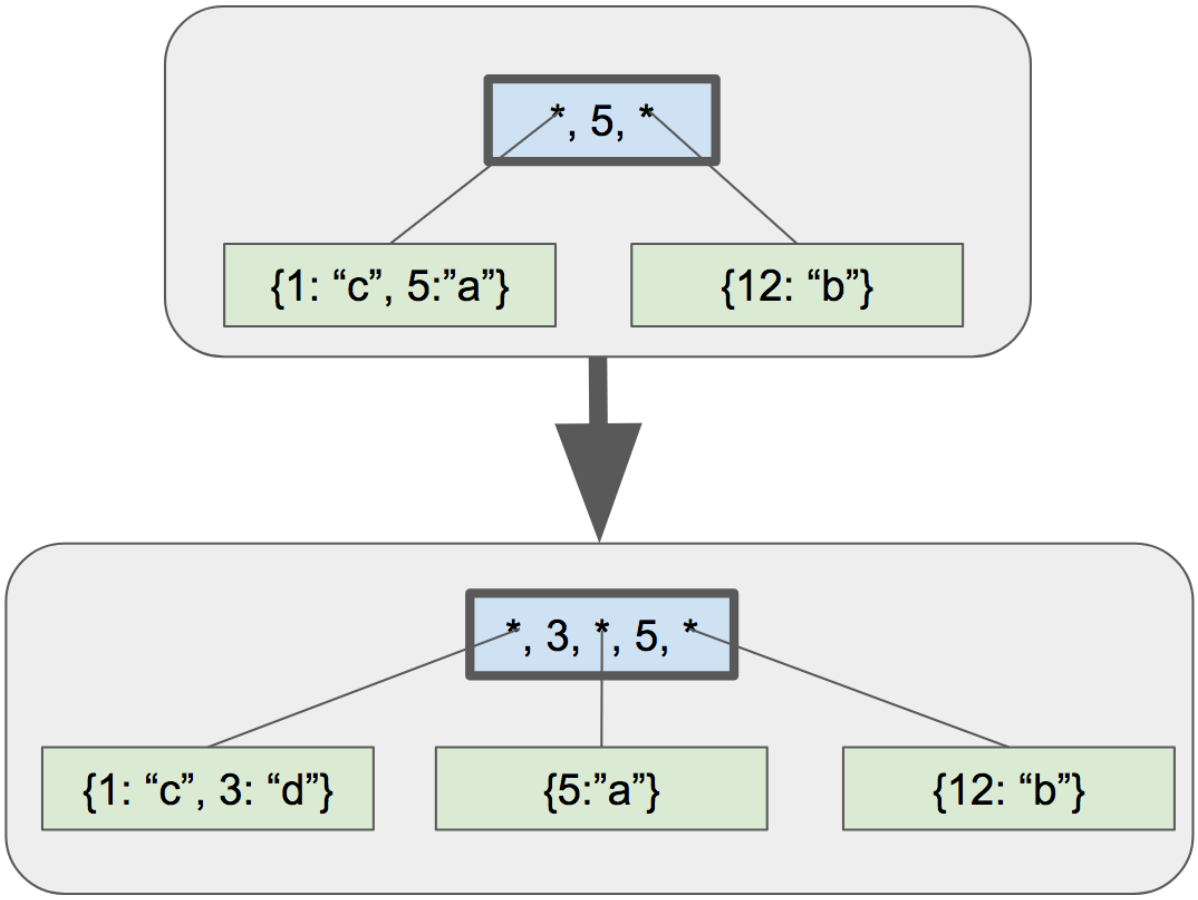
Overview

View on GitHub (pull requests welcome)

Part 13 - Updating Parent Node After a Split

< Part 12 - Scanning a Multi-Level B-Tree

For the next step on our epic b-tree implementation journey, we're going to handle fixing up the parent node after splitting a leaf. I'm going to use the following example as a reference:



Example of updating internal node

In this example, we add the key "3" to the tree. That causes the left leaf node to split. After the split we fix up the tree by doing the following:

1. Update the first key in the parent to be the maximum key in the left child ("3")
2. Add a new child pointer / key pair after the updated key
    - The new pointer points to the new child node
    - The new key is the maximum key in the new child node ("5")

So first things first, replace our stub code with two new function calls: `update_internal_node_key()` for step 1 and `internal_node_insert()` for step 2

```
@@ -670,9 +725,11 @@ void leaf_node_split_and_insert(Cursor* cur
   */

   void* old_node = get_page(cursor->table->pager, cursor->page_
+  uint32_t old_max = get_node_max_key(old_node);
   uint32_t new_page_num = get_unused_page_num(cursor->table->pa
   void* new_node = get_page(cursor->table->pager, new_page_num)
   initialize_leaf_node(new_node);
+  *node_parent(new_node) = *node_parent(old_node);
   *leaf_node_next_leaf(new_node) = *leaf_node_next_leaf(old_noc
   *leaf_node_next_leaf(old_node) = new_page_num;

@@ -709,8 +766,12 @@ void leaf_node_split_and_insert(Cursor* cur
   if (is_node_root(old_node)) {
     return create_new_root(cursor->table, new_page_num);
   } else {
-    printf("Need to implement updating parent after split\n");
-    exit(EXIT_FAILURE);
+    uint32_t parent_page_num = *node_parent(old_node);
+    uint32_t new_max = get_node_max_key(old_node);
+    void* parent = get_page(cursor->table->pager, parent_page_n
+
+    update_internal_node_key(parent, old_max, new_max);
+    internal_node_insert(cursor->table, parent_page_num, new_pa
+    return;
   }
 }
```

In order to get a reference to the parent, we need to start recording in each node a pointer to its parent node.

```
+uint32_t* node_parent(void* node) { return node + PARENT_POINTE
```

```
@@ -660,6 +675,48 @@ void create_new_root(Table* table, uint32_t
   uint32_t left_child_max_key = get_node_max_key(left_child);
   *internal_node_key(root, 0) = left_child_max_key;
   *internal_node_right_child(root) = right_child_page_num;
+  *node_parent(left_child) = table->root_page_num;
+  *node_parent(right_child) = table->root_page_num;
 }
```

Now we need to find the affected cell in the parent node. The child doesn't know its own page number, so we can't look for that. But it does know its own maximum key, so we can search the parent for that key.

```
+void update_internal_node_key(void* node, uint32_t old_key, uin
+  uint32_t old_child_index = internal_node_find_child(node, old
+  *internal_node_key(node, old_child_index) = new_key;
 }
```

Inside `internal_node_find_child()` we'll reuse some code we already have for finding a key in an internal node. Refactor `internal_node_find()` to use the new helper method.

```
-Cursor* internal_node_find(Table* table, uint32_t page_num, uin
-  void* node = get_page(table->pager, page_num);
+uint32_t internal_node_find_child(void* node, uint32_t key) {
+  /*
+  Return the index of the child which should contain
+  the given key.
+  */
+
   uint32_t num_keys = *internal_node_num_keys(node);

-  /* Binary search to find index of child to search */
+  /* Binary search */
```

```
   uint32_t min_index = 0;
   uint32_t max_index = num_keys; /* there is one more child tha

@@ -386,7 +394,14 @@ Cursor* internal_node_find(Table* table, ui
     }
   }

-  uint32_t child_num = *internal_node_child(node, min_index);
+  return min_index;
+}
+
+Cursor* internal_node_find(Table* table, uint32_t page_num, uin
+  void* node = get_page(table->pager, page_num);
+
+  uint32_t child_index = internal_node_find_child(node, key);
+  uint32_t child_num = *internal_node_child(node, child_index);
   void* child = get_page(table->pager, child_num);
   switch (get_node_type(child)) {
     case NODE_LEAF:
```

Now we get to the heart of this article, implementing `internal_node_insert()`. I'll explain it in pieces.

```
+void internal_node_insert(Table* table, uint32_t parent_page_nu
+                          uint32_t child_page_num) {
+  /*
+  Add a new child/key pair to parent that corresponds to child
+  */
+
+  void* parent = get_page(table->pager, parent_page_num);
+  void* child = get_page(table->pager, child_page_num);
+  uint32_t child_max_key = get_node_max_key(child);
+  uint32_t index = internal_node_find_child(parent, child_max_k
+
+  uint32_t original_num_keys = *internal_node_num_keys(parent);
+  *internal_node_num_keys(parent) = original_num_keys + 1;
+
+  if (original_num_keys >= INTERNAL_NODE_MAX_CELLS) {
+    printf("Need to implement splitting internal node\n");
```

```
+    exit(EXIT_FAILURE);
+  }
```

The index where the new cell (child/key pair) should be inserted depends on the maximum key in the new child. In the example we looked at, `child_max_key` would be 5 and `index` would be 1.

If there's no room in the internal node for another cell, throw an error. We'll implement that later.

Now let's look at the rest of the function:

```
+
+  uint32_t right_child_page_num = *internal_node_right_child(pa
+  void* right_child = get_page(table->pager, right_child_page_r
+
+  if (child_max_key > get_node_max_key(right_child)) {
+    /* Replace right child */
+    *internal_node_child(parent, original_num_keys) = right_chi
+    *internal_node_key(parent, original_num_keys) =
+        get_node_max_key(right_child);
+    *internal_node_right_child(parent) = child_page_num;
+  } else {
+    /* Make room for the new cell */
+    for (uint32_t i = original_num_keys; i > index; i--) {
+      void* destination = internal_node_cell(parent, i);
+      void* source = internal_node_cell(parent, i - 1);
+      memcpy(destination, source, INTERNAL_NODE_CELL_SIZE);
+    }
+    *internal_node_child(parent, index) = child_page_num;
+    *internal_node_key(parent, index) = child_max_key;
+  }
+}
```

Because we store the rightmost child pointer separately from the rest of the child/key pairs, we have to handle things differently if the new child is going to become the rightmost child.

In our example, we would get into the `else` block. First we make room for the new

cell by shifting other cells one space to the right. (Although in our example there are 0 cells to shift)

Next, we write the new child pointer and key into the cell determined by `index`.

To reduce the size of testcases needed, I'm hardcoding `INTERNAL_NODE_MAX_CELLS` for now

```
@@ -126,6 +126,8 @@ const uint32_t INTERNAL_NODE_KEY_SIZE = size
 const uint32_t INTERNAL_NODE_CHILD_SIZE = sizeof(uint32_t);
 const uint32_t INTERNAL_NODE_CELL_SIZE =
     INTERNAL_NODE_CHILD_SIZE + INTERNAL_NODE_KEY_SIZE;
+/* Keep this small for testing */
+const uint32_t INTERNAL_NODE_MAX_CELLS = 3;
```

Speaking of tests, our large-dataset test gets past our old stub and gets to our new one:

```
@@ -65,7 +65,7 @@ describe 'database' do
     result = run_script(script)
     expect(result.last(2)).to match_array([
       "db > Executed.",
-      "db > Need to implement updating parent after split",
+      "db > Need to implement splitting internal node",
     ])
```

Very satisfying, I know.

I'll add another test that prints a four-node tree. Just so we test more cases than sequential ids, this test will add records in a pseudorandom order.

```
+  it 'allows printing out the structure of a 4-leaf-node btree'
+    script = [
+      "insert 18 user18 person18@example.com",
+      "insert 7 user7 person7@example.com",
+      "insert 10 user10 person10@example.com",
+      "insert 29 user29 person29@example.com",
+      "insert 23 user23 person23@example.com",
+      "insert 4 user4 person4@example.com",
```

```
+      "insert 14 user14 person14@example.com",
+      "insert 30 user30 person30@example.com",
+      "insert 15 user15 person15@example.com",
+      "insert 26 user26 person26@example.com",
+      "insert 22 user22 person22@example.com",
+      "insert 19 user19 person19@example.com",
+      "insert 2 user2 person2@example.com",
+      "insert 1 user1 person1@example.com",
+      "insert 21 user21 person21@example.com",
+      "insert 11 user11 person11@example.com",
+      "insert 6 user6 person6@example.com",
+      "insert 20 user20 person20@example.com",
+      "insert 5 user5 person5@example.com",
+      "insert 8 user8 person8@example.com",
+      "insert 9 user9 person9@example.com",
+      "insert 3 user3 person3@example.com",
+      "insert 12 user12 person12@example.com",
+      "insert 27 user27 person27@example.com",
+      "insert 17 user17 person17@example.com",
+      "insert 16 user16 person16@example.com",
+      "insert 13 user13 person13@example.com",
+      "insert 24 user24 person24@example.com",
+      "insert 25 user25 person25@example.com",
+      "insert 28 user28 person28@example.com",
+      ".btree",
+      ".exit",
+    ]
+    result = run_script(script)
```

As-is, it will output this:

```
- internal (size 3)
  - leaf (size 7)
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
```

```
    - 7
  - key 1
  - leaf (size 8)
    - 8
    - 9
    - 10
    - 11
    - 12
    - 13
    - 14
    - 15
  - key 15
  - leaf (size 7)
    - 16
    - 17
    - 18
    - 19
    - 20
    - 21
    - 22
  - key 22
  - leaf (size 8)
    - 23
    - 24
    - 25
    - 26
    - 27
    - 28
    - 29
    - 30
db >
```

Look carefully and you'll spot a bug:

```
    - 5
    - 6
    - 7
  - key 1
```

The key there should be 7, not 1!

After a bunch of debugging, I discovered this was due to some bad pointer arithmetic.

```
 uint32_t* internal_node_key(void* node, uint32_t key_num) {
-  return internal_node_cell(node, key_num) + INTERNAL_NODE_CHIL
+  return (void*)internal_node_cell(node, key_num) + INTERNAL_NO
 }
```

`INTERNAL_NODE_CHILD_SIZE` is 4. My intention here was to add 4 bytes to the result of `internal_node_cell()`, but since `internal_node_cell()` returns a `uint32_t*`, this it was actually adding `4 * sizeof(uint32_t)` bytes. I fixed it by casting to a `void*` before doing the arithmetic.

NOTE! Pointer arithmetic on void pointers is not part of the C standard and may not work with your compiler. I may do an article in the future on portability, but I'm leaving my void pointer arithmetic for now.

Alright. One more step toward a fully-operational btree implementation. The next step should be splitting internal nodes. Until then!

< Part 12 - Scanning a Multi-Level B-Tree
