Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 12 - Scanning a Multi-Level B-Tree

< Part 11 - Recursively Searching the B-Tree

Part 13 - Updating Parent Node After a Split >

We now support constructing a multi-level btree, but we've broken `select` statements in the process. Here's a test case that inserts 15 rows and then tries to print them.

```
+  it 'prints all rows in a multi-level tree' do
+    script = []
+    (1..15).each do |i|
+      script << "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << "select"
+    script << ".exit"
+    result = run_script(script)
+
+    expect(result[15...result.length]).to match_array([
+      "db > (1, user1, person1@example.com)",
+      "(2, user2, person2@example.com)",
+      "(3, user3, person3@example.com)",
+      "(4, user4, person4@example.com)",
+      "(5, user5, person5@example.com)",
+      "(6, user6, person6@example.com)",
+      "(7, user7, person7@example.com)",
+      "(8, user8, person8@example.com)",
+      "(9, user9, person9@example.com)",
```

```
+      "(10, user10, person10@example.com)",
+      "(11, user11, person11@example.com)",
+      "(12, user12, person12@example.com)",
+      "(13, user13, person13@example.com)",
+      "(14, user14, person14@example.com)",
+      "(15, user15, person15@example.com)",
+      "Executed.", "db > ",
+    ])
+  end
```

But when we run that test case right now, what actually happens is:

```
db > select
(2, user1, person1@example.com)
Executed.
```

That's weird. It's only printing one row, and that row looks corrupted (notice the id doesn't match the username).

The weirdness is because `execute_select()` begins at the start of the table, and our current implementation of `table_start()` returns cell 0 of the root node. But the root of our tree is now an internal node which doesn't contain any rows. The data that was printed must have been left over from when the root node was a leaf. `execute_select()` should really return cell 0 of the leftmost leaf node.

So get rid of the old implementation:

```
-Cursor* table_start(Table* table) {
-  Cursor* cursor = malloc(sizeof(Cursor));
-  cursor->table = table;
-  cursor->page_num = table->root_page_num;
-  cursor->cell_num = 0;
-
-  void* root_node = get_page(table->pager, table->root_page_num
-  uint32_t num_cells = *leaf_node_num_cells(root_node);
-  cursor->end_of_table = (num_cells == 0);
-
-  return cursor;
-}
```

And add a new implementation that searches for key 0 (the minimum possible key). Even if key 0 does not exist in the table, this method will return the position of the lowest id (the start of the left-most leaf node).

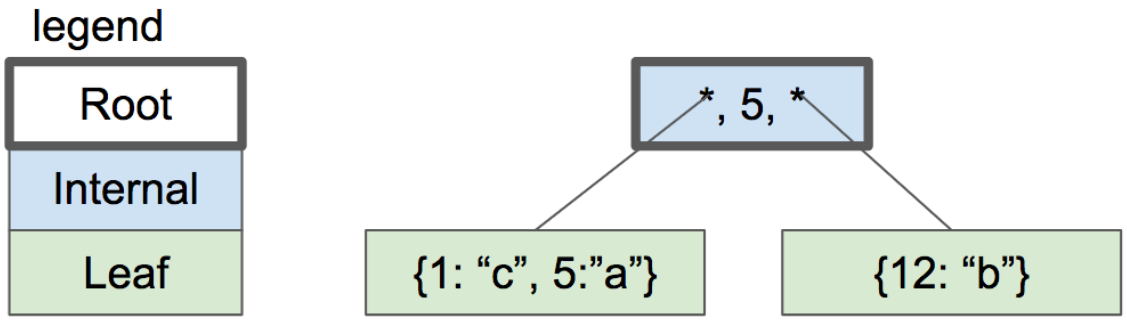
```
+Cursor* table_start(Table* table) {
+  Cursor* cursor =  table_find(table, 0);
+
+  void* node = get_page(table->pager, cursor->page_num);
+  uint32_t num_cells = *leaf_node_num_cells(node);
+  cursor->end_of_table = (num_cells == 0);
+
+  return cursor;
+}
```

With those changes, it still only prints out one node's worth of rows:

```
db > select
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
Executed.
```

```
db >
```

With 15 entries, our btree consists of one internal node and two leaf nodes, which looks something like this:



structure of our btree

To scan the entire table, we need to jump to the second leaf node after we reach the end of the first. To do that, we're going to save a new field in the leaf node header called "next_leaf", which will hold the page number of the leaf's sibling node on the right. The rightmost leaf node will have a `next_leaf` value of 0 to denote no sibling (page 0 is reserved for the root node of the table anyway).

Update the leaf node header format to include the new field:

```
 const uint32_t LEAF_NODE_NUM_CELLS_SIZE = sizeof(uint32_t);
 const uint32_t LEAF_NODE_NUM_CELLS_OFFSET = COMMON_NODE_HEADER_
-const uint32_t LEAF_NODE_HEADER_SIZE =
-    COMMON_NODE_HEADER_SIZE + LEAF_NODE_NUM_CELLS_SIZE;
+const uint32_t LEAF_NODE_NEXT_LEAF_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_NEXT_LEAF_OFFSET =
+    LEAF_NODE_NUM_CELLS_OFFSET + LEAF_NODE_NUM_CELLS_SIZE;
+const uint32_t LEAF_NODE_HEADER_SIZE = COMMON_NODE_HEADER_SIZE
+                                       LEAF_NODE_NUM_CELLS_SIZE
+                                       LEAF_NODE_NEXT_LEAF_SIZE
```

Add a method to access the new field:

```
+uint32_t* leaf_node_next_leaf(void* node) {
```

```
+  return node + LEAF_NODE_NEXT_LEAF_OFFSET;
+}
```

Set `next_leaf` to 0 by default when initializing a new leaf node:

```
@@ -322,6 +330,7 @@ void initialize_leaf_node(void* node) {
   set_node_type(node, NODE_LEAF);
   set_node_root(node, false);
   *leaf_node_num_cells(node) = 0;
+  *leaf_node_next_leaf(node) = 0;  // 0 represents no sibling
 }
```

Whenever we split a leaf node, update the sibling pointers. The old leaf's sibling becomes the new leaf, and the new leaf's sibling becomes whatever used to be the old leaf's sibling.

```
@@ -659,6 +671,8 @@ void leaf_node_split_and_insert(Cursor* curs
   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);
+  *leaf_node_next_leaf(new_node) = *leaf_node_next_leaf(old_noc
+  *leaf_node_next_leaf(old_node) = new_page_num;
```

Adding a new field changes a few constants:

```
   it 'prints constants' do
     script = [
       ".constants",
@@ -199,9 +228,9 @@ describe 'database' do
       "db > Constants:",
       "ROW_SIZE: 293",
       "COMMON_NODE_HEADER_SIZE: 6",
-      "LEAF_NODE_HEADER_SIZE: 10",
+      "LEAF_NODE_HEADER_SIZE: 14",
       "LEAF_NODE_CELL_SIZE: 297",
-      "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+      "LEAF_NODE_SPACE_FOR_CELLS: 4082",
       "LEAF_NODE_MAX_CELLS: 13",
```

```
        "db > ",
      ])
```

Now whenever we want to advance the cursor past the end of a leaf node, we can check if the leaf node has a sibling. If it does, jump to it. Otherwise, we're at the end of the table.

```
@@ -428,7 +432,15 @@ void cursor_advance(Cursor* cursor) {

   cursor->cell_num += 1;
   if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
-    cursor->end_of_table = true;
+    /* Advance to next leaf node */
+    uint32_t next_page_num = *leaf_node_next_leaf(node);
+    if (next_page_num == 0) {
+      /* This was rightmost leaf */
+      cursor->end_of_table = true;
+    } else {
+      cursor->page_num = next_page_num;
+      cursor->cell_num = 0;
+    }
   }
 }
```

After those changes, we actually print 15 rows…

```
db > select
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
(8, user8, person8@example.com)
(9, user9, person9@example.com)
(10, user10, person10@example.com)
(11, user11, person11@example.com)
(12, user12, person12@example.com)
```

```
(13, user13, person13@example.com)
(1919251317, 14, on14@example.com)
(15, user15, person15@example.com)
Executed.
db >
```

…but one of them looks corrupted

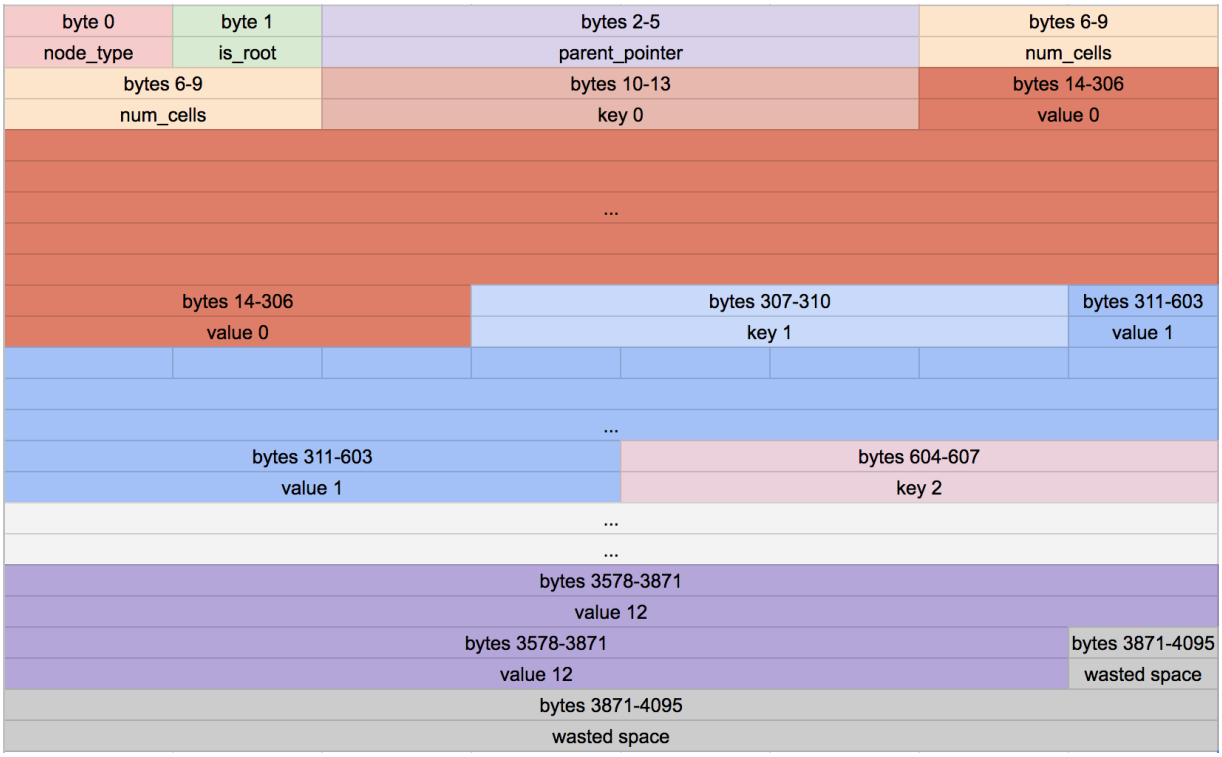
```
(1919251317, 14, on14@example.com)
```

After some debugging, I found out it's because of a bug in how we split leaf nodes:

```
@@ -676,7 +690,9 @@ void leaf_node_split_and_insert(Cursor* curs
     void* destination = leaf_node_cell(destination_node, index_

     if (i == cursor->cell_num) {
-      serialize_row(value, destination);
+      serialize_row(value,
+                    leaf_node_value(destination_node, index_wi
+      *leaf_node_key(destination_node, index_within_node) = key
     } else if (i > cursor->cell_num) {
       memcpy(destination, leaf_node_cell(old_node, i - 1), LEAF
     } else {
```

Remember that each cell in a leaf node consists of first a key then a value:



Original leaf node format

We were writing the new row (value) into the start of the cell, where the key should go. That means part of the username was going into the section for id (hence the crazy large id).

After fixing that bug, we finally print out the entire table as expected:

```
db > select
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
(8, user8, person8@example.com)
(9, user9, person9@example.com)
(10, user10, person10@example.com)
(11, user11, person11@example.com)
(12, user12, person12@example.com)
(13, user13, person13@example.com)
(14, user14, person14@example.com)
```

```
(15, user15, person15@example.com)
Executed.
db >
```

Whew! One bug after another, but we're making progress.

Until next time.

< Part 11 - Recursively Searching the B-Tree

Part 13 - Updating Parent Node After a Split >
