Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

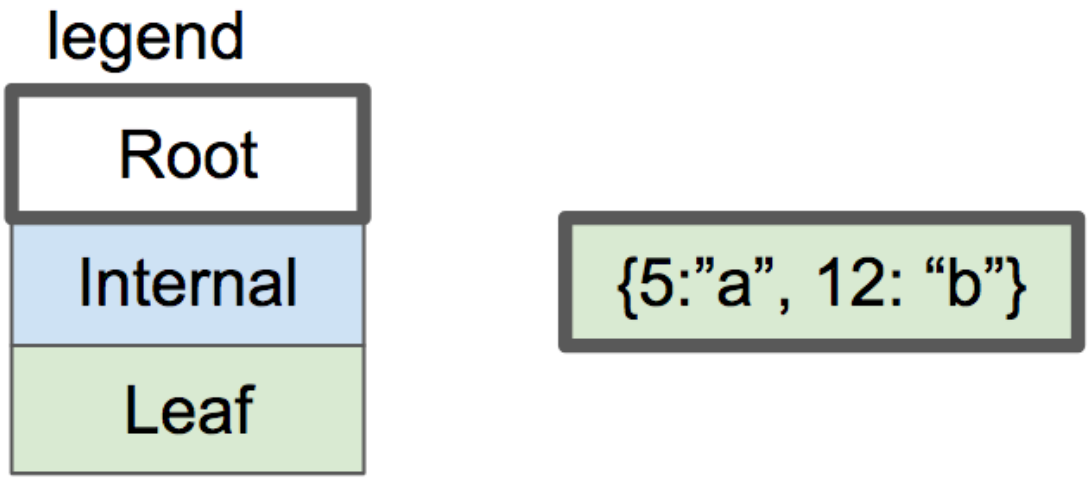
Part 10 - Splitting a Leaf Node

< Part 9 - Binary Search and Duplicate Keys

Part 11 - Recursively Searching the B-Tree >

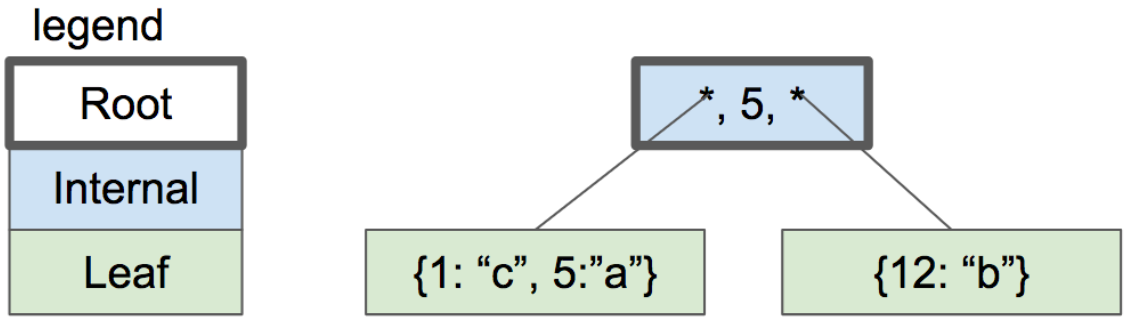
Our B-Tree doesn't feel like much of a tree with only one node. To fix that, we need some code to split a leaf node in twain. And after that, we need to create an internal node to serve as a parent for the two leaf nodes.

Basically our goal for this article is to go from this:



one-node btree

to this:



two-level btree

First things first, let's remove the error handling for a full leaf node:

```
 void leaf_node_insert(Cursor* cursor, uint32_t key, Row* value)
   void* node = get_page(cursor->table->pager, cursor->page_num)

   uint32_t num_cells = *leaf_node_num_cells(node);
   if (num_cells >= LEAF_NODE_MAX_CELLS) {
     // Node full
-    printf("Need to implement splitting a leaf node.\n");
-    exit(EXIT_FAILURE);
+    leaf_node_split_and_insert(cursor, key, value);
+    return;
   }
```

```
ExecuteResult execute_insert(Statement* statement, Table* table)
   void* node = get_page(table->pager, table->root_page_num);
   uint32_t num_cells = (*leaf_node_num_cells(node));
-  if (num_cells >= LEAF_NODE_MAX_CELLS) {
-    return EXECUTE_TABLE_FULL;
-  }

   Row* row_to_insert = &(statement->row_to_insert);
   uint32_t key_to_insert = row_to_insert->id;
```

Splitting Algorithm

Easy part's over. Here's a description of what we need to do from SQLite Database

System: Design and Implementation

> *If there is no space on the leaf node, we would split the existing entries residing there and the new one (being inserted) into two equal halves: lower and upper halves. (Keys on the upper half are strictly greater than those on the lower half.) We allocate a new leaf node, and move the upper half into the new node.*

Let's get a handle to the old node and create the new node:

```
+void leaf_node_split_and_insert(Cursor* cursor, uint32_t key, F
+  /*
+  Create a new node and move half the cells over.
+  Insert the new value in one of the two nodes.
+  Update parent or create a new parent.
+  */
+
+  void* old_node = get_page(cursor->table->pager, cursor->page_
+  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);
```

Next, copy every cell into its new location:

```
+  /*
+  All existing keys plus new key should be divided
+  evenly between old (left) and new (right) nodes.
+  Starting from the right, move each key to correct position.
+  */
+  for (int32_t i = LEAF_NODE_MAX_CELLS; i >= 0; i--) {
+    void* destination_node;
+    if (i >= LEAF_NODE_LEFT_SPLIT_COUNT) {
+      destination_node = new_node;
+    } else {
+      destination_node = old_node;
+    }
+    uint32_t index_within_node = i % LEAF_NODE_LEFT_SPLIT_COUNT
+    void* destination = leaf_node_cell(destination_node, index_
+
+    if (i == cursor->cell_num) {
```

```
+      serialize_row(value, destination);
+    } else if (i > cursor->cell_num) {
+      memcpy(destination, leaf_node_cell(old_node, i - 1), LEAF
+    } else {
+      memcpy(destination, leaf_node_cell(old_node, i), LEAF_NOD
+    }
+  }
```

Update cell counts in each node's header:

```
+  /* Update cell count on both leaf nodes */
+  *(leaf_node_num_cells(old_node)) = LEAF_NODE_LEFT_SPLIT_COUNT
+  *(leaf_node_num_cells(new_node)) = LEAF_NODE_RIGHT_SPLIT_COUN
```

Then we need to update the nodes' parent. If the original node was the root, it had no parent. In that case, create a new root node to act as the parent. I'll stub out the other branch for now:

```
+  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);
+  }
+}
```

Allocating New Pages

Let's go back and define a few new functions and constants. When we created a new leaf node, we put it in a page decided by `get_unused_page_num()`:

```
+/*
+Until we start recycling free pages, new pages will always
+go onto the end of the database file
+*/
+uint32_t get_unused_page_num(Pager* pager) { return pager->num_
```

For now, we're assuming that in a database with N pages, page numbers 0 through

N-1 are allocated. Therefore we can always allocate page number N for new pages. Eventually after we implement deletion, some pages may become empty and their page numbers unused. To be more efficient, we could re-allocate those free pages.

Leaf Node Sizes

To keep the tree balanced, we evenly distribute cells between the two new nodes. If a leaf node can hold N cells, then during a split we need to distribute N+1 cells between two nodes (N original cells plus one new one). I'm arbitrarily choosing the left node to get one more cell if N+1 is odd.

```
+const uint32_t LEAF_NODE_RIGHT_SPLIT_COUNT = (LEAF_NODE_MAX_CEL
+const uint32_t LEAF_NODE_LEFT_SPLIT_COUNT =
+    (LEAF_NODE_MAX_CELLS + 1) - LEAF_NODE_RIGHT_SPLIT_COUNT;
```

Creating a New Root

Here's how SQLite Database System explains the process of creating a new root node:

> *Let N be the root node. First allocate two nodes, say L and R. Move lower half of N into L and the upper half into R. Now N is empty. Add ⟨L, K,R⟩ in N, where K is the max key in L. Page N remains the root. Note that the depth of the tree has increased by one, but the new tree remains height balanced without violating any B+-tree property.*

At this point, we've already allocated the right child and moved the upper half into it. Our function takes the right child as input and allocates a new page to store the left child.

```
+void create_new_root(Table* table, uint32_t right_child_page_nu
+  /*
+  Handle splitting the root.
+  Old root copied to new page, becomes left child.
+  Address of right child passed in.
+  Re-initialize root page to contain the new root node.
+  New root node points to two children.
+  */
+
+  void* root = get_page(table->pager, table->root_page_num);
```

```
+  void* right_child = get_page(table->pager, right_child_page_r
+  uint32_t left_child_page_num = get_unused_page_num(table->pag
+  void* left_child = get_page(table->pager, left_child_page_num
```

The old root is copied to the left child so we can reuse the root page:

```
+  /* Left child has data copied from old root */
+  memcpy(left_child, root, PAGE_SIZE);
+  set_node_root(left_child, false);
```

Finally we initialize the root page as a new internal node with two children.

```
+  /* Root node is a new internal node with one key and two chil
+  initialize_internal_node(root);
+  set_node_root(root, true);
+  *internal_node_num_keys(root) = 1;
+  *internal_node_child(root, 0) = left_child_page_num;
+  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;
+}
```

Internal Node Format

Now that we're finally creating an internal node, we have to define its layout. It starts with the common header, then the number of keys it contains, then the page number of its rightmost child. Internal nodes always have one more child pointer than they have keys. That extra child pointer is stored in the header.

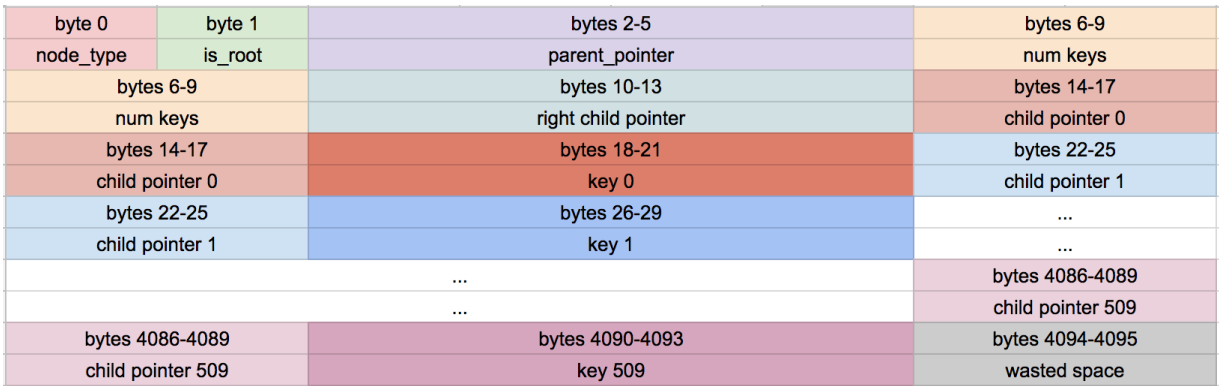
```
+/*
+ * Internal Node Header Layout
+ */
+const uint32_t INTERNAL_NODE_NUM_KEYS_SIZE = sizeof(uint32_t);
+const uint32_t INTERNAL_NODE_NUM_KEYS_OFFSET = COMMON_NODE_HEAD
+const uint32_t INTERNAL_NODE_RIGHT_CHILD_SIZE = sizeof(uint32_t
+const uint32_t INTERNAL_NODE_RIGHT_CHILD_OFFSET =
+    INTERNAL_NODE_NUM_KEYS_OFFSET + INTERNAL_NODE_NUM_KEYS_SIZE
+const uint32_t INTERNAL_NODE_HEADER_SIZE = COMMON_NODE_HEADER_S
```

```
+                                               INTERNAL_NODE_NUM_KE
+                                               INTERNAL_NODE_RIGHT_
```

The body is an array of cells where each cell contains a child pointer and a key. Every key should be the maximum key contained in the child to its left.

```
+/*
+ * Internal Node Body Layout
+ */
+const uint32_t INTERNAL_NODE_KEY_SIZE = sizeof(uint32_t);
+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;
```

Based on these constants, here's how the layout of an internal node will look:



Our internal node format

Notice our huge branching factor. Because each child pointer / key pair is so small, we can fit 510 keys and 511 child pointers in each internal node. That means we'll never have to traverse many layers of the tree to find a given key!

# internal node layers	max # leaf nodes	Size of all leaf nodes
0	511^0 = 1	4 KB
1	511^1 = 512	~2 MB
2	511^2 = 261,121	~1 GB
3	511^3 = 133,432,831	~550 GB

In actuality, we can't store a full 4 KB of data per leaf node due to the overhead of

the header, keys, and wasted space. But we can search through something like 500 GB of data by loading only 4 pages from disk. This is why the B-Tree is a useful data structure for databases.

Here are the methods for reading and writing to an internal node:

```
+uint32_t* internal_node_num_keys(void* node) {
+  return node + INTERNAL_NODE_NUM_KEYS_OFFSET;
+}
+
+uint32_t* internal_node_right_child(void* node) {
+  return node + INTERNAL_NODE_RIGHT_CHILD_OFFSET;
+}
+
+uint32_t* internal_node_cell(void* node, uint32_t cell_num) {
+  return node + INTERNAL_NODE_HEADER_SIZE + cell_num * INTERNAL
+}
+
+uint32_t* internal_node_child(void* node, uint32_t child_num) {
+  uint32_t num_keys = *internal_node_num_keys(node);
+  if (child_num > num_keys) {
+    printf("Tried to access child_num %d > num_keys %d\n", chil
+    exit(EXIT_FAILURE);
+  } else if (child_num == num_keys) {
+    return internal_node_right_child(node);
+  } else {
+    return internal_node_cell(node, child_num);
+  }
+}
+
+uint32_t* internal_node_key(void* node, uint32_t key_num) {
+  return internal_node_cell(node, key_num) + INTERNAL_NODE_CHIL
+}
```

For an internal node, the maximum key is always its right key. For a leaf node, it's the key at the maximum index:

```
+uint32_t get_node_max_key(void* node) {
+  switch (get_node_type(node)) {
```

```
+    case NODE_INTERNAL:
+      return *internal_node_key(node, *internal_node_num_keys(r
+    case NODE_LEAF:
+      return *leaf_node_key(node, *leaf_node_num_cells(node) -
+  }
+}
```

Keeping Track of the Root

We're finally using the `is_root` field in the common node header. Recall that we use it to decide how to split a leaf node:

```
  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);
  }
}
```

Here are the getter and setter:

```
+bool is_node_root(void* node) {
+  uint8_t value = *((uint8_t*)(node + IS_ROOT_OFFSET));
+  return (bool)value;
+}
+
+void set_node_root(void* node, bool is_root) {
+  uint8_t value = is_root;
+  *((uint8_t*)(node + IS_ROOT_OFFSET)) = value;
+}
```

Initializing both types of nodes should default to setting `is_root` to false:

```
 void initialize_leaf_node(void* node) {
   set_node_type(node, NODE_LEAF);
+  set_node_root(node, false);
   *leaf_node_num_cells(node) = 0;
```

```
 }

+void initialize_internal_node(void* node) {
+  set_node_type(node, NODE_INTERNAL);
+  set_node_root(node, false);
+  *internal_node_num_keys(node) = 0;
+}
```

We should set `is_root` to true when creating the first node of the table:

```
     // New database file. Initialize page 0 as leaf node.
     void* root_node = get_page(pager, 0);
     initialize_leaf_node(root_node);
+    set_node_root(root_node, true);
   }

   return table;
```

Printing the Tree

To help us visualize the state of the database, we should update our `.btree` metacommand to print a multi-level tree.

I'm going to replace the current `print_leaf_node()` function

```
-void print_leaf_node(void* node) {
-  uint32_t num_cells = *leaf_node_num_cells(node);
-  printf("leaf (size %d)\n", num_cells);
-  for (uint32_t i = 0; i < num_cells; i++) {
-    uint32_t key = *leaf_node_key(node, i);
-    printf("  - %d : %d\n", i, key);
-  }
-}
```

with a new recursive function that takes any node, then prints it and its children. It takes an indentation level as a parameter, which increases with each recursive call. I'm also adding a tiny helper function to indent.

```
+void indent(uint32_t level) {
+  for (uint32_t i = 0; i < level; i++) {
+    printf("  ");
+  }
+}
+
+void print_tree(Pager* pager, uint32_t page_num, uint32_t inder
+  void* node = get_page(pager, page_num);
+  uint32_t num_keys, child;
+
+  switch (get_node_type(node)) {
+    case (NODE_LEAF):
+      num_keys = *leaf_node_num_cells(node);
+      indent(indentation_level);
+      printf("- leaf (size %d)\n", num_keys);
+      for (uint32_t i = 0; i < num_keys; i++) {
+        indent(indentation_level + 1);
+        printf("- %d\n", *leaf_node_key(node, i));
+      }
+      break;
+    case (NODE_INTERNAL):
+      num_keys = *internal_node_num_keys(node);
+      indent(indentation_level);
+      printf("- internal (size %d)\n", num_keys);
+      for (uint32_t i = 0; i < num_keys; i++) {
+        child = *internal_node_child(node, i);
+        print_tree(pager, child, indentation_level + 1);
+
+        indent(indentation_level + 1);
+        printf("- key %d\n", *internal_node_key(node, i));
+      }
+      child = *internal_node_right_child(node);
+      print_tree(pager, child, indentation_level + 1);
+      break;
+  }
+}
```

And update the call to the print function, passing an indentation level of zero.

```
   } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
     printf("Tree:\n");
-    print_leaf_node(get_page(table->pager, 0));
+    print_tree(table->pager, 0, 0);
     return META_COMMAND_SUCCESS;
```

Here's a test case for the new printing functionality!

```
+  it 'allows printing out the structure of a 3-leaf-node btree'
+    script = (1..14).map do |i|
+      "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << ".btree"
+    script << "insert 15 user15 person15@example.com"
+    script << ".exit"
+    result = run_script(script)
+
+    expect(result[14...(result.length)]).to match_array([
+      "db > Tree:",
+      "- internal (size 1)",
+      "  - leaf (size 7)",
+      "    - 1",
+      "    - 2",
+      "    - 3",
+      "    - 4",
+      "    - 5",
+      "    - 6",
+      "    - 7",
+      "  - key 7",
+      "  - leaf (size 7)",
+      "    - 8",
+      "    - 9",
+      "    - 10",
+      "    - 11",
+      "    - 12",
+      "    - 13",
+      "    - 14",
+      "db > Need to implement searching an internal node",
+    ])
```

```
+  end
```

The new format is a little simplified, so we need to update the existing `.btree` test:

```
       "db > Executed.",
       "db > Executed.",
       "db > Tree:",
-      "leaf (size 3)",
-      "  - 0 : 1",
-      "  - 1 : 2",
-      "  - 2 : 3",
+      "- leaf (size 3)",
+      "  - 1",
+      "  - 2",
+      "  - 3",
       "db > "
     ])
   end
```

Here's the `.btree` output of the new test on its own:

```
Tree:
- internal (size 1)
  - leaf (size 7)
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
  - key 7
  - leaf (size 7)
    - 8
    - 9
    - 10
    - 11
    - 12
    - 13
```

```
    - 14
```

On the least indented level, we see the root node (an internal node). It says `size 1` because it has one key. Indented one level, we see a leaf node, a key, and another leaf node. The key in the root node (7) is is the maximum key in the first leaf node. Every key greater than 7 is in the second leaf node.

A Major Problem

If you've been following along closely you may notice we've missed something big. Look what happens if we try to insert one additional row:

```
db > insert 15 user15 person15@example.com
Need to implement searching an internal node
```

Whoops! Who wrote that TODO message? :P

Next time we'll continue the epic B-tree saga by implementing search on a multi-level tree.

< Part 9 - Binary Search and Duplicate Keys

Part 11 - Recursively Searching the B-Tree >
