Let's Build a Simple Database

Writing a sqlite clone from scratch in C

Overview

View on GitHub (pull requests welcome)

Part 8 - B-Tree Leaf Node Format

< Part 7 - Introduction to the B-Tree

Part 9 - Binary Search and Duplicate Keys >

We're changing the format of our table from an unsorted array of rows to a B-Tree. This is a pretty big change that is going to take multiple articles to implement. By the end of this article, we'll define the layout of a leaf node and support inserting key/value pairs into a single-node tree. But first, let's recap the reasons for switching to a tree structure.

Alternative Table Formats

With the current format, each page stores only rows (no metadata) so it is pretty space efficient. Insertion is also fast because we just append to the end. However, finding a particular row can only be done by scanning the entire table. And if we want to delete a row, we have to fill in the hole by moving every row that comes after it.

If we stored the table as an array, but kept rows sorted by id, we could use binary search to find a particular id. However, insertion would be slow because we would have to move a lot of rows to make space.

Instead, we're going with a tree structure. Each node in the tree can contain a variable number of rows, so we have to store some information in each node to keep track of how many rows it contains. Plus there is the storage overhead of all the internal nodes which don't store any rows. In exchange for a larger database file, we get fast insertion, deletion and lookup.

	Unsorted Array of rows	Sorted Array of rows	Tree of nodes
Pages contain	only data	only data	metadata, primary keys, and data
Rows per page	more	more	fewer
Insertion	O(1)	O(n)	O(log(n))
Deletion	O(n)	O(n)	O(log(n))
Lookup by id	O(n)	O(log(n))	O(log(n))

Node Header Format

Leaf nodes and internal nodes have different layouts. Let's make an enum to keep track of node type:

```
+typedef enum { NODE_INTERNAL, NODE_LEAF } NodeType;
```

Each node will correspond to one page. Internal nodes will point to their children by storing the page number that stores the child. The btree asks the pager for a particular page number and gets back a pointer into the page cache. Pages are stored in the database file one after the other in order of page number.

Nodes need to store some metadata in a header at the beginning of the page. Every node will store what type of node it is, whether or not it is the root node, and a pointer to its parent (to allow finding a node's siblings). I define constants for the size and offset of every header field:

```
+/*
+ * Common Node Header Layout
+ */
+const uint32_t NODE_TYPE_SIZE = sizeof(uint8_t);
+const uint32_t NODE_TYPE_OFFSET = 0;
+const uint32_t IS_ROOT_SIZE = sizeof(uint8_t);
+const uint32_t IS_ROOT_OFFSET = NODE_TYPE_SIZE;
+const uint32_t PARENT_POINTER_SIZE = sizeof(uint32_t);
+const uint32_t PARENT_POINTER_OFFSET = IS_ROOT_OFFSET + IS_ROOT
```

```
+const uint8_t COMMON_NODE_HEADER_SIZE =
+    NODE_TYPE_SIZE + IS_ROOT_SIZE + PARENT_POINTER_SIZE;
```

Leaf Node Format

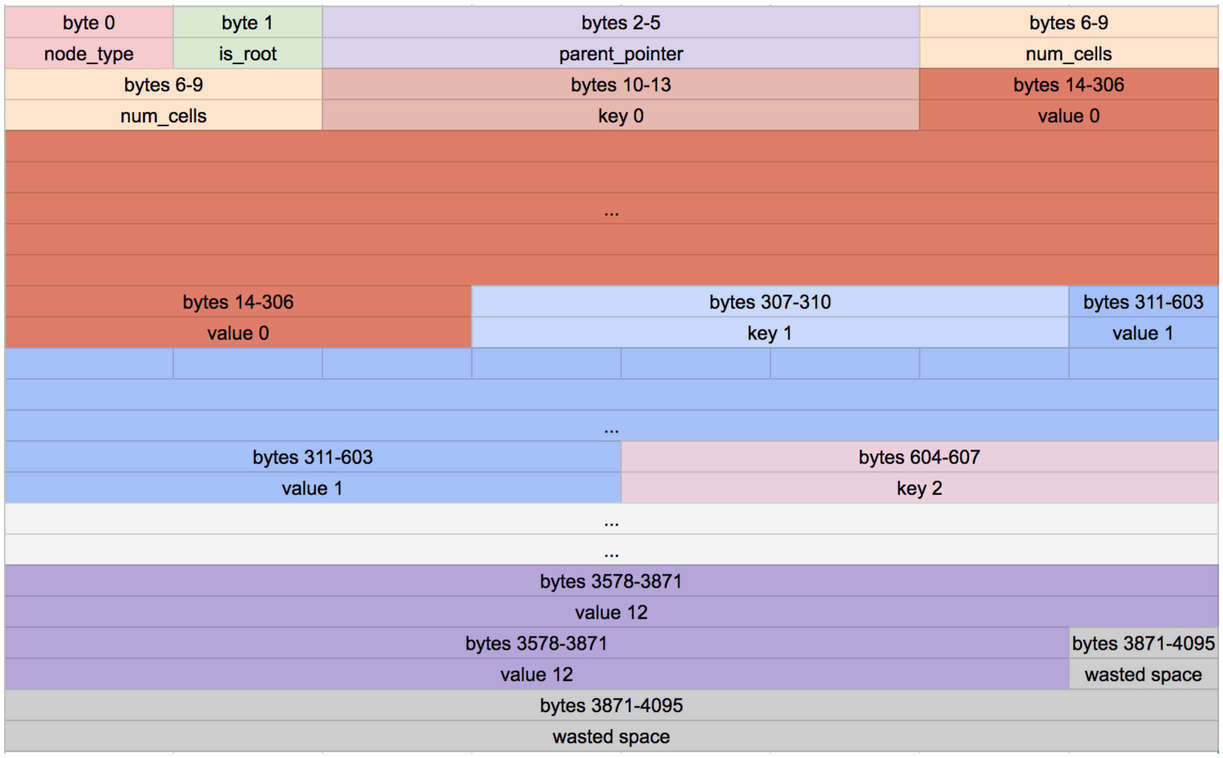
In addition to these common header fields, leaf nodes need to store how many "cells" they contain. A cell is a key/value pair.

```
+/*
+ * Leaf Node Header Layout
+ */
+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;
```

The body of a leaf node is an array of cells. Each cell is a key followed by a value (a serialized row).

```
+/*
+ * Leaf Node Body Layout
+ */
+const uint32_t LEAF_NODE_KEY_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_KEY_OFFSET = 0;
+const uint32_t LEAF_NODE_VALUE_SIZE = ROW_SIZE;
+const uint32_t LEAF_NODE_VALUE_OFFSET =
+    LEAF_NODE_KEY_OFFSET + LEAF_NODE_KEY_SIZE;
+const uint32_t LEAF_NODE_CELL_SIZE = LEAF_NODE_KEY_SIZE + LEAF_
+const uint32_t LEAF_NODE_SPACE_FOR_CELLS = PAGE_SIZE - LEAF_NOD
+const uint32_t LEAF_NODE_MAX_CELLS =
+    LEAF_NODE_SPACE_FOR_CELLS / LEAF_NODE_CELL_SIZE;
```

Based on these constants, here's what the layout of a leaf node looks like currently:



Our leaf node format

It's a little space inefficient to use an entire byte per boolean value in the header, but this makes it easier to write code to access those values.

Also notice that there's some wasted space at the end. We store as many cells as we can after the header, but the leftover space can't hold an entire cell. We leave it empty to avoid splitting cells between nodes.

Accessing Leaf Node Fields

The code to access keys, values and metadata all involve pointer arithmetic using the constants we just defined.

```
+uint32_t* leaf_node_num_cells(void* node) {
+  return node + LEAF_NODE_NUM_CELLS_OFFSET;
+}
+
+void* leaf_node_cell(void* node, uint32_t cell_num) {
+  return node + LEAF_NODE_HEADER_SIZE + cell_num * LEAF_NODE_CE
+}
+
+uint32_t* leaf_node_key(void* node, uint32_t cell_num) {
+  return leaf_node_cell(node, cell_num);
```

```
+}
+
+void* leaf_node_value(void* node, uint32_t cell_num) {
+  return leaf_node_cell(node, cell_num) + LEAF_NODE_KEY_SIZE;
+}
+
+void initialize_leaf_node(void* node) { *leaf_node_num_cells(nc
+
```

These methods return a pointer to the value in question, so they can be used both as a getter and a setter.

Changes to Pager and Table Objects

Every node is going to take up exactly one page, even if it's not full. That means our pager no longer needs to support reading/writing partial pages.

```
-void pager_flush(Pager* pager, uint32_t page_num, uint32_t size
+void pager_flush(Pager* pager, uint32_t page_num) {
   if (pager->pages[page_num] == NULL) {
     printf("Tried to flush null page\n");
     exit(EXIT_FAILURE);
@@ -242,7 +337,7 @@ void pager_flush(Pager* pager, uint32_t page
   }

   ssize_t bytes_written =
-      write(pager->file_descriptor, pager->pages[page_num], siz
+      write(pager->file_descriptor, pager->pages[page_num], PAC

   if (bytes_written == -1) {
     printf("Error writing: %d\n", errno);
```

```
 void db_close(Table* table) {
   Pager* pager = table->pager;
-  uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;

-  for (uint32_t i = 0; i < num_full_pages; i++) {
+  for (uint32_t i = 0; i < pager->num_pages; i++) {
     if (pager->pages[i] == NULL) {
       continue;
     }
-    pager_flush(pager, i, PAGE_SIZE);
+    pager_flush(pager, i);
     free(pager->pages[i]);
     pager->pages[i] = NULL;
   }

-  // There may be a partial page to write to the end of the fi
-  // This should not be needed after we switch to a B-tree
-  uint32_t num_additional_rows = table->num_rows % ROWS_PER_PA
-  if (num_additional_rows > 0) {
-    uint32_t page_num = num_full_pages;
-    if (pager->pages[page_num] != NULL) {
-      pager_flush(pager, page_num, num_additional_rows * ROW_S
-      free(pager->pages[page_num]);
-      pager->pages[page_num] = NULL;
-    }
-  }
-
   int result = close(pager->file_descriptor);
   if (result == -1) {
     printf("Error closing db file.\n");
```

Now it makes more sense to store the number of pages in our database rather than the number of rows. The number of pages should be associated with the pager object, not the table, since it's the number of pages used by the database, not a particular table. A btree is identified by its root node page number, so the table object needs to keep track of that.

```
 const uint32_t PAGE_SIZE = 4096;
 const uint32_t TABLE_MAX_PAGES = 100;
-const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
-const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES

 typedef struct {
   int file_descriptor;
   uint32_t file_length;
+  uint32_t num_pages;
```

```
   void* pages[TABLE_MAX_PAGES];
 } Pager;

 typedef struct {
   Pager* pager;
-  uint32_t num_rows;
+  uint32_t root_page_num;
 } Table;
```

```
@@ -127,6 +200,10 @@ void* get_page(Pager* pager, uint32_t page_
     }

     pager->pages[page_num] = page;
+
+    if (page_num >= pager->num_pages) {
+      pager->num_pages = page_num + 1;
+    }
   }

   return pager->pages[page_num];
```

```
@@ -184,6 +269,12 @@ Pager* pager_open(const char* filename) {
   Pager* pager = malloc(sizeof(Pager));
   pager->file_descriptor = fd;
   pager->file_length = file_length;
+  pager->num_pages = (file_length / PAGE_SIZE);
+
+  if (file_length % PAGE_SIZE != 0) {
+    printf("Db file is not a whole number of pages. Corrupt fi
+    exit(EXIT_FAILURE);
+  }

   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
     pager->pages[i] = NULL;
```

Changes to the Cursor Object

A cursor represents a position in the table. When our table was a simple array of rows, we could access a row given just the row number. Now that it's a tree, we identify a position by the page number of the node, and the cell number within that node.

```
 typedef struct {
   Table* table;
-  uint32_t row_num;
+  uint32_t page_num;
+  uint32_t cell_num;
   bool end_of_table;  // Indicates a position one past the last
 } Cursor;
```

```
 Cursor* table_start(Table* table) {
   Cursor* cursor = malloc(sizeof(Cursor));
   cursor->table = table;
-  cursor->row_num = 0;
-  cursor->end_of_table = (table->num_rows == 0);
+  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;
 }
```

```
 Cursor* table_end(Table* table) {
   Cursor* cursor = malloc(sizeof(Cursor));
   cursor->table = table;
-  cursor->row_num = table->num_rows;
+  cursor->page_num = table->root_page_num;
+
+  void* root_node = get_page(table->pager, table->root_page_num
+  uint32_t num_cells = *leaf_node_num_cells(root_node);
+  cursor->cell_num = num_cells;
   cursor->end_of_table = true;
```

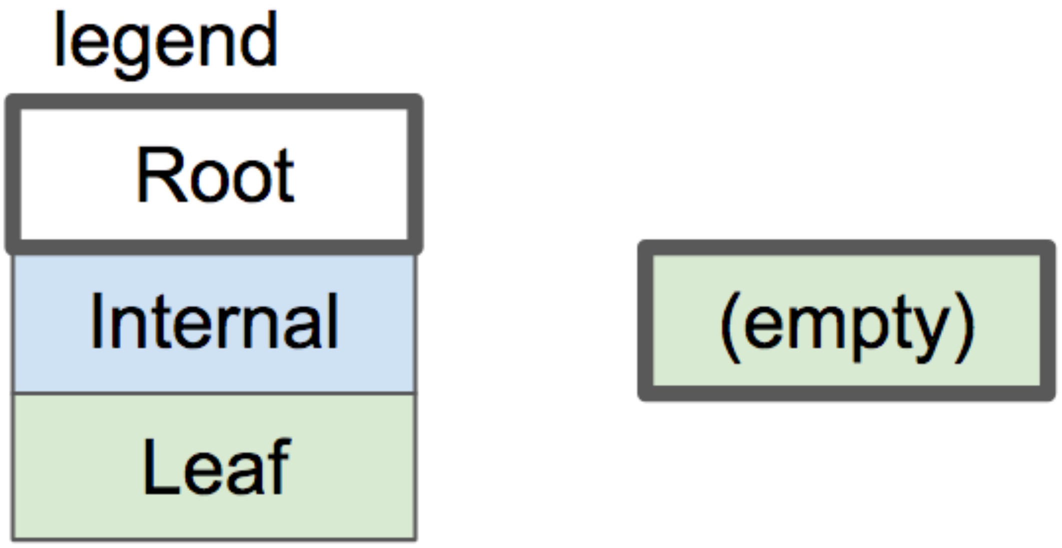
```diff
   return cursor;
 }
```

```diff
 void* cursor_value(Cursor* cursor) {
-  uint32_t row_num = cursor->row_num;
-  uint32_t page_num = row_num / ROWS_PER_PAGE;
+  uint32_t page_num = cursor->page_num;
   void* page = get_page(cursor->table->pager, page_num);
-  uint32_t row_offset = row_num % ROWS_PER_PAGE;
-  uint32_t byte_offset = row_offset * ROW_SIZE;
-  return page + byte_offset;
+  return leaf_node_value(page, cursor->cell_num);
 }
```

```diff
 void cursor_advance(Cursor* cursor) {
-  cursor->row_num += 1;
-  if (cursor->row_num >= cursor->table->num_rows) {
+  uint32_t page_num = cursor->page_num;
+  void* node = get_page(cursor->table->pager, page_num);
+
+  cursor->cell_num += 1;
+  if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
     cursor->end_of_table = true;
   }
 }
```

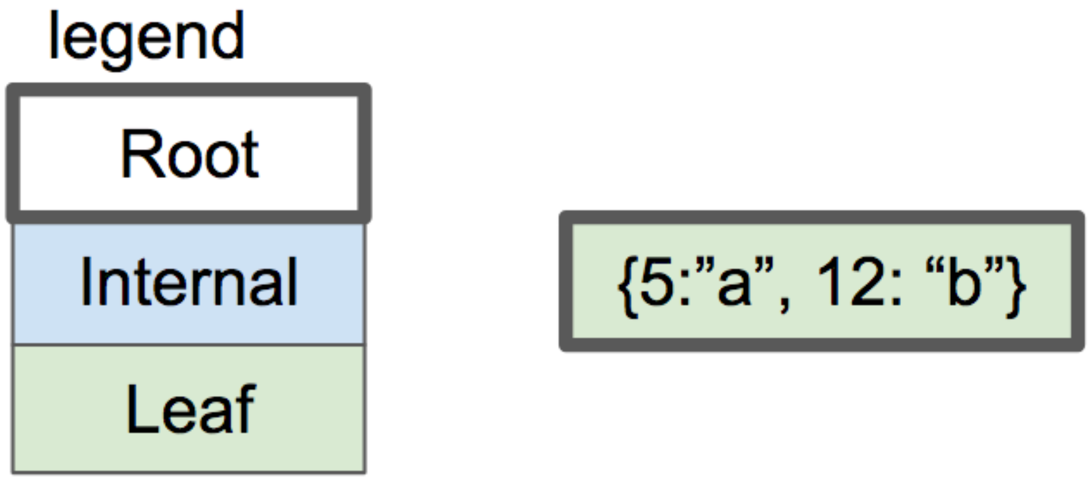
Insertion Into a Leaf Node

In this article we're only going to implement enough to get a single-node tree. Recall from last article that a tree starts out as an empty leaf node:



empty btree

Key/value pairs can be added until the leaf node is full:



one-node btree

When we open the database for the first time, the database file will be empty, so we initialize page 0 to be an empty leaf node (the root node):

```
 Table* db_open(const char* filename) {
   Pager* pager = pager_open(filename);
-  uint32_t num_rows = pager->file_length / ROW_SIZE;
```

```
   Table* table = malloc(sizeof(Table));
   table->pager = pager;
-  table->num_rows = num_rows;
+  table->root_page_num = 0;
+
+  if (pager->num_pages == 0) {
+    // New database file. Initialize page 0 as leaf node.
+    void* root_node = get_page(pager, 0);
+    initialize_leaf_node(root_node);
+  }

   return table;
 }
```

Next we’ll make a function for inserting a key/value pair into a leaf node. It will take a cursor as input to represent the position where the pair should be inserted.

```
+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);
+  }
+
+  if (cursor->cell_num < num_cells) {
+    // Make room for new cell
+    for (uint32_t i = num_cells; i > cursor->cell_num; i--) {
+      memcpy(leaf_node_cell(node, i), leaf_node_cell(node, i -
+             LEAF_NODE_CELL_SIZE);
+    }
+  }
+
+  *(leaf_node_num_cells(node)) += 1;
+  *(leaf_node_key(node, cursor->cell_num)) = key;
+  serialize_row(value, leaf_node_value(node, cursor->cell_num))
+}
+
```

We haven't implemented splitting yet, so we error if the node is full. Next we shift cells one space to the right to make room for the new cell. Then we write the new key/value into the empty space.

Since we assume the tree only has one node, our `execute_insert()` function simply needs to call this helper method:

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

   Row* row_to_insert = &(statement->row_to_insert);
   Cursor* cursor = table_end(table);

-  serialize_row(row_to_insert, cursor_value(cursor));
-  table->num_rows += 1;
+  leaf_node_insert(cursor, row_to_insert->id, row_to_insert);

   free(cursor);
```

With those changes, our database should work as before! Except now it returns a "Table Full" error much sooner, since we can't split the root node yet.

How many rows can the leaf node hold?

Command to Print Constants

I'm adding a new meta command to print out a few constants of interest.

```
+void print_constants() {
+  printf("ROW_SIZE: %d\n", ROW_SIZE);
+  printf("COMMON_NODE_HEADER_SIZE: %d\n", COMMON_NODE_HEADER_SI
+  printf("LEAF_NODE_HEADER_SIZE: %d\n", LEAF_NODE_HEADER_SIZE);
+  printf("LEAF_NODE_CELL_SIZE: %d\n", LEAF_NODE_CELL_SIZE);
+  printf("LEAF_NODE_SPACE_FOR_CELLS: %d\n", LEAF_NODE_SPACE_FOF
+  printf("LEAF_NODE_MAX_CELLS: %d\n", LEAF_NODE_MAX_CELLS);
+}
+
@@ -294,6 +376,14 @@ MetaCommandResult do_meta_command(InputBuf
   if (strcmp(input_buffer->buffer, ".exit") == 0) {
     db_close(table);
     exit(EXIT_SUCCESS);
+  } else if (strcmp(input_buffer->buffer, ".constants") == 0) {
+    printf("Constants:\n");
+    print_constants();
+    return META_COMMAND_SUCCESS;
   } else {
     return META_COMMAND_UNRECOGNIZED_COMMAND;
   }
```

I'm also adding a test so we get alerted when those constants change:

```
+  it 'prints constants' do
+    script = [
+      ".constants",
+      ".exit",
+    ]
+    result = run_script(script)
+
+    expect(result).to match_array([
+      "db > Constants:",
+      "ROW_SIZE: 293",
+      "COMMON_NODE_HEADER_SIZE: 6",
+      "LEAF_NODE_HEADER_SIZE: 10",
+      "LEAF_NODE_CELL_SIZE: 297",
+      "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+      "LEAF_NODE_MAX_CELLS: 13",
+      "db > ",
+    ])
+  end
```

So our table can hold 13 rows right now!

Tree Visualization

To help with debugging and visualization, I’m also adding a meta command to print out a representation of the btree.

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

```
@@ -294,6 +376,14 @@ MetaCommandResult do_meta_command(InputBuff
   if (strcmp(input_buffer->buffer, ".exit") == 0) {
     db_close(table);
     exit(EXIT_SUCCESS);
+  } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
+    printf("Tree:\n");
+    print_leaf_node(get_page(table->pager, 0));
+    return META_COMMAND_SUCCESS;
   } else if (strcmp(input_buffer->buffer, ".constants") == 0) {
     printf("Constants:\n");
     print_constants();
     return META_COMMAND_SUCCESS;
   } else {
     return META_COMMAND_UNRECOGNIZED_COMMAND;
   }
```

And a test

```
+  it 'allows printing out the structure of a one-node btree' do
+    script = [3, 1, 2].map do |i|
+      "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << ".btree"
+    script << ".exit"
+    result = run_script(script)
+
+    expect(result).to match_array([
+      "db > Executed.",
+      "db > Executed.",
+      "db > Executed.",
+      "db > Tree:",
+      "leaf (size 3)",
+      "  - 0 : 3",
+      "  - 1 : 1",
+      "  - 2 : 2",
+      "db > "
+    ])
+  end
```

Uh oh, we're still not storing rows in sorted order. You'll notice that `execute_insert()` inserts into the leaf node at the position returned by `table_end()`. So rows are stored in the order they were inserted, just like before.

Next Time

This all might seem like a step backwards. Our database now stores fewer rows than it did before, and we're still storing rows in unsorted order. But like I said at the beginning, this is a big change and it's important to break it up into manageable steps.

Next time, we'll implement finding a record by primary key, and start storing rows in sorted order.

Complete Diff

```
@@ -62,29 +62,101 @@ const uint32_t ROW_SIZE = ID_SIZE + USERNAM

 const uint32_t PAGE_SIZE = 4096;
 #define TABLE_MAX_PAGES 100
-const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
-const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES
```

```
 typedef struct {
   int file_descriptor;
   uint32_t file_length;
+  uint32_t num_pages;
   void* pages[TABLE_MAX_PAGES];
 } Pager;

 typedef struct {
   Pager* pager;
-  uint32_t num_rows;
+  uint32_t root_page_num;
 } Table;

 typedef struct {
   Table* table;
-  uint32_t row_num;
+  uint32_t page_num;
+  uint32_t cell_num;
   bool end_of_table;  // Indicates a position one past the last
 } Cursor;

+typedef enum { NODE_INTERNAL, NODE_LEAF } NodeType;
+
+/*
+ * Common Node Header Layout
+ */
+const uint32_t NODE_TYPE_SIZE = sizeof(uint8_t);
+const uint32_t NODE_TYPE_OFFSET = 0;
+const uint32_t IS_ROOT_SIZE = sizeof(uint8_t);
+const uint32_t IS_ROOT_OFFSET = NODE_TYPE_SIZE;
+const uint32_t PARENT_POINTER_SIZE = sizeof(uint32_t);
+const uint32_t PARENT_POINTER_OFFSET = IS_ROOT_OFFSET + IS_ROOT
+const uint8_t COMMON_NODE_HEADER_SIZE =
+    NODE_TYPE_SIZE + IS_ROOT_SIZE + PARENT_POINTER_SIZE;
+
+/*
+ * Leaf Node Header Layout
+ */
+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;
+
+/*
+ * Leaf Node Body Layout
+ */
+const uint32_t LEAF_NODE_KEY_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_KEY_OFFSET = 0;
+const uint32_t LEAF_NODE_VALUE_SIZE = ROW_SIZE;
+const uint32_t LEAF_NODE_VALUE_OFFSET =
+    LEAF_NODE_KEY_OFFSET + LEAF_NODE_KEY_SIZE;
+const uint32_t LEAF_NODE_CELL_SIZE = LEAF_NODE_KEY_SIZE + LEAF_
+const uint32_t LEAF_NODE_SPACE_FOR_CELLS = PAGE_SIZE - LEAF_NOD
+const uint32_t LEAF_NODE_MAX_CELLS =
+    LEAF_NODE_SPACE_FOR_CELLS / LEAF_NODE_CELL_SIZE;
+
+uint32_t* leaf_node_num_cells(void* node) {
+  return node + LEAF_NODE_NUM_CELLS_OFFSET;
+}
+
+void* leaf_node_cell(void* node, uint32_t cell_num) {
+  return node + LEAF_NODE_HEADER_SIZE + cell_num * LEAF_NODE_CE
+}
+
+uint32_t* leaf_node_key(void* node, uint32_t cell_num) {
+  return leaf_node_cell(node, cell_num);
+}
+
+void* leaf_node_value(void* node, uint32_t cell_num) {
+  return leaf_node_cell(node, cell_num) + LEAF_NODE_KEY_SIZE;
+}
+
+void print_constants() {
+  printf("ROW_SIZE: %d\n", ROW_SIZE);
+  printf("COMMON_NODE_HEADER_SIZE: %d\n", COMMON_NODE_HEADER_SI
+  printf("LEAF_NODE_HEADER_SIZE: %d\n", LEAF_NODE_HEADER_SIZE);
+  printf("LEAF_NODE_CELL_SIZE: %d\n", LEAF_NODE_CELL_SIZE);
+  printf("LEAF_NODE_SPACE_FOR_CELLS: %d\n", LEAF_NODE_SPACE_FOR
+  printf("LEAF_NODE_MAX_CELLS: %d\n", LEAF_NODE_MAX_CELLS);
+}
```

```
+
+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);
+  }
+}
+
 void print_row(Row* row) {
     printf("(%d, %s, %s)\n", row->id, row->username, row->emai
 }
@@ -101,6 +173,8 @@ void deserialize_row(void *source, Row* dest
     memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_
 }

+void initialize_leaf_node(void* node) { *leaf_node_num_cells(nc
+
 void* get_page(Pager* pager, uint32_t page_num) {
   if (page_num > TABLE_MAX_PAGES) {
     printf("Tried to fetch page number out of bounds. %d > %d\r
@@ -128,6 +202,10 @@ void* get_page(Pager* pager, uint32_t page_
     }

     pager->pages[page_num] = page;
+
+    if (page_num >= pager->num_pages) {
+      pager->num_pages = page_num + 1;
+    }
   }

   return pager->pages[page_num];
@@ -136,8 +214,12 @@ void* get_page(Pager* pager, uint32_t page_
 Cursor* table_start(Table* table) {
   Cursor* cursor = malloc(sizeof(Cursor));
   cursor->table = table;
-  cursor->row_num = 0;
-  cursor->end_of_table = (table->num_rows == 0);
+  cursor->page_num = table->root_page_num;
+  cursor->cell_num = 0;
+
+  void* root_node = get_page(table->pager, table->root_page_nun
+  uint32_t num_cells = *leaf_node_num_cells(root_node);
+  cursor->end_of_table = (num_cells == 0);

   return cursor;
 }
@@ -145,24 +227,28 @@ Cursor* table_start(Table* table) {
 Cursor* table_end(Table* table) {
   Cursor* cursor = malloc(sizeof(Cursor));
   cursor->table = table;
-  cursor->row_num = table->num_rows;
+  cursor->page_num = table->root_page_num;
+
+  void* root_node = get_page(table->pager, table->root_page_nun
+  uint32_t num_cells = *leaf_node_num_cells(root_node);
+  cursor->cell_num = num_cells;
   cursor->end_of_table = true;

   return cursor;
 }

 void* cursor_value(Cursor* cursor) {
-  uint32_t row_num = cursor->row_num;
-  uint32_t page_num = row_num / ROWS_PER_PAGE;
+  uint32_t page_num = cursor->page_num;
   void* page = get_page(cursor->table->pager, page_num);
-  uint32_t row_offset = row_num % ROWS_PER_PAGE;
-  uint32_t byte_offset = row_offset * ROW_SIZE;
-  return page + byte_offset;
+  return leaf_node_value(page, cursor->cell_num);
 }

 void cursor_advance(Cursor* cursor) {
-  cursor->row_num += 1;
-  if (cursor->row_num >= cursor->table->num_rows) {
+  uint32_t page_num = cursor->page_num;
+  void* node = get_page(cursor->table->pager, page_num);
+
+  cursor->cell_num += 1;
+  if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
     cursor->end_of_table = true;
   }
 }
@@ -185,6 +271,12 @@ Pager* pager_open(const char* filename) {
   Pager* pager = malloc(sizeof(Pager));
   pager->file_descriptor = fd;
   pager->file_length = file_length;
+  pager->num_pages = (file_length / PAGE_SIZE);
+
+  if (file_length % PAGE_SIZE != 0) {
+    printf("Db file is not a whole number of pages. Corrupt fi
+    exit(EXIT_FAILURE);
+  }

   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
     pager->pages[i] = NULL;
@@ -194,11 +285,15 @@ Pager* pager_open(const char* filename) {
@@ -195,11 +287,16 @@ Pager* pager_open(const char* filename) {

 Table* db_open(const char* filename) {
   Pager* pager = pager_open(filename);
-  uint32_t num_rows = pager->file_length / ROW_SIZE;

   Table* table = malloc(sizeof(Table));
   table->pager = pager;
-  table->num_rows = num_rows;
+  table->root_page_num = 0;
+
+  if (pager->num_pages == 0) {
+    // New database file. Initialize page 0 as leaf node.
+    void* root_node = get_page(pager, 0);
+    initialize_leaf_node(root_node);
+  }

   return table;
 }
@@ -234,7 +331,7 @@ void close_input_buffer(InputBuffer* input_b
     free(input_buffer);
 }

-void pager_flush(Pager* pager, uint32_t page_num, uint32_t size
+void pager_flush(Pager* pager, uint32_t page_num) {
   if (pager->pages[page_num] == NULL) {
     printf("Tried to flush null page\n");
     exit(EXIT_FAILURE);
@@ -242,7 +337,7 @@ void pager_flush(Pager* pager, uint32_t page
@@ -249,7 +346,7 @@ void pager_flush(Pager* pager, uint32_t page
   }

   ssize_t bytes_written =
-      write(pager->file_descriptor, pager->pages[page_num], siz
+      write(pager->file_descriptor, pager->pages[page_num], PAG

   if (bytes_written == -1) {
     printf("Error writing: %d\n", errno);
@@ -252,29 +347,16 @@ void pager_flush(Pager* pager, uint32_t pa
@@ -260,29 +357,16 @@ void pager_flush(Pager* pager, uint32_t pa

 void db_close(Table* table) {
   Pager* pager = table->pager;
-  uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;

-  for (uint32_t i = 0; i < num_full_pages; i++) {
+  for (uint32_t i = 0; i < pager->num_pages; i++) {
     if (pager->pages[i] == NULL) {
       continue;
     }
-    pager_flush(pager, i, PAGE_SIZE);
+    pager_flush(pager, i);
     free(pager->pages[i]);
     pager->pages[i] = NULL;
   }

-  // There may be a partial page to write to the end of the fil
-  // This should not be needed after we switch to a B-tree
-  uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAG
-  if (num_additional_rows > 0) {
-    uint32_t page_num = num_full_pages;
-    if (pager->pages[page_num] != NULL) {
-      pager_flush(pager, page_num, num_additional_rows * ROW_SI
-      free(pager->pages[page_num]);
-      pager->pages[page_num] = NULL;
-    }
-  }
-
   int result = close(pager->file_descriptor);
   if (result == -1) {
     printf("Error closing db file.\n");
@@ -305,6 +389,14 @@ MetaCommandResult do_meta_command(InputBuf
   if (strcmp(input_buffer->buffer, ".exit") == 0) {
     db_close(table);
     exit(EXIT_SUCCESS);
+  } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
+    printf("Tree:\n");
+    print_leaf_node(get_page(table->pager, 0));
+    return META_COMMAND_SUCCESS;
+  } else if (strcmp(input_buffer->buffer, ".constants") == 0) {
+    printf("Constants:\n");
+    print_constants();
+    return META_COMMAND_SUCCESS;
   } else {
     return META_COMMAND_UNRECOGNIZED_COMMAND;
   }
@@ -354,16 +446,39 @@ PrepareResult prepare_statement(InputBuffe
   return PREPARE_UNRECOGNIZED_STATEMENT;
 }

+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);
+  }
+
+  if (cursor->cell_num < num_cells) {
```

```
+    // Make room for new cell
+    for (uint32_t i = num_cells; i > cursor->cell_num; i--) {
+      memcpy(leaf_node_cell(node, i), leaf_node_cell(node, i -
+             LEAF_NODE_CELL_SIZE);
+    }
+  }
+
+  *(leaf_node_num_cells(node)) += 1;
+  *(leaf_node_key(node, cursor->cell_num)) = key;
+  serialize_row(value, leaf_node_value(node, cursor->cell_num)
+}
+
 ExecuteResult execute_insert(Statement* statement, Table* table
-  if (table->num_rows >= TABLE_MAX_ROWS) {
+  void* node = get_page(table->pager, table->root_page_num);
+  if ((*leaf_node_num_cells(node) >= LEAF_NODE_MAX_CELLS)) {
     return EXECUTE_TABLE_FULL;
   }

   Row* row_to_insert = &(statement->row_to_insert);
   Cursor* cursor = table_end(table);

-  serialize_row(row_to_insert, cursor_value(cursor));
-  table->num_rows += 1;
+  leaf_node_insert(cursor, row_to_insert->id, row_to_insert);

   free(cursor);
```

And the specs:

```
+  it 'allows printing out the structure of a one-node btree' do
+    script = [3, 1, 2].map do |i|
+      "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << ".btree"
+    script << ".exit"
+    result = run_script(script)
+
+    expect(result).to match_array([
+       "db > Executed.",
+       "db > Executed.",
+       "db > Executed.",
+       "db > Tree:",
+       "leaf (size 3)",
+       "  - 0 : 3",
+       "  - 1 : 1",
+       "  - 2 : 2",
+       "db > "
+     ])
+   end
+
+   it 'prints constants' do
+     script = [
+       ".constants",
+       ".exit",
+     ]
+     result = run_script(script)
+
+     expect(result).to match_array([
+       "db > Constants:",
+       "ROW_SIZE: 293",
+       "COMMON_NODE_HEADER_SIZE: 6",
+       "LEAF_NODE_HEADER_SIZE: 10",
+       "LEAF_NODE_CELL_SIZE: 297",
+       "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+       "LEAF_NODE_MAX_CELLS: 13",
+       "db > ",
+     ])
+   end
  end
```

< Part 7 - Introduction to the B-Tree

Part 9 - Binary Search and Duplicate Keys >

rss | subscribe by email