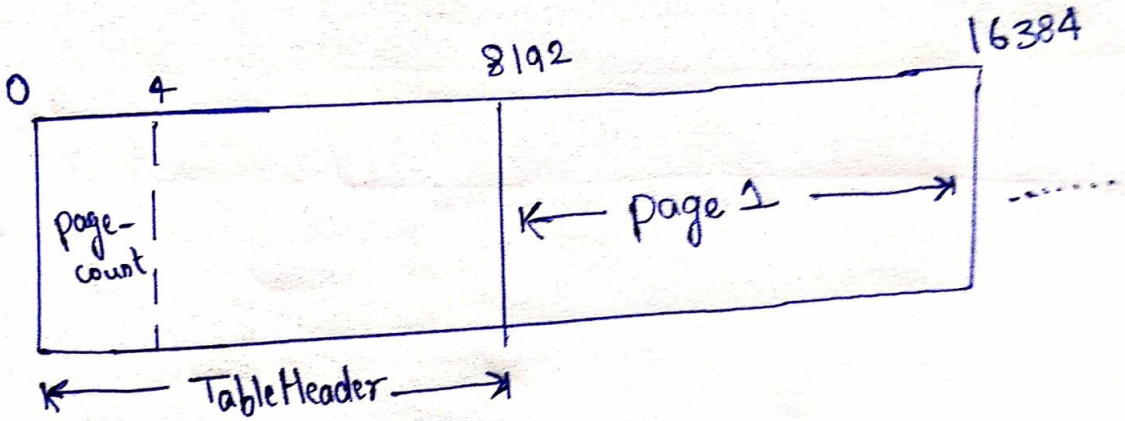
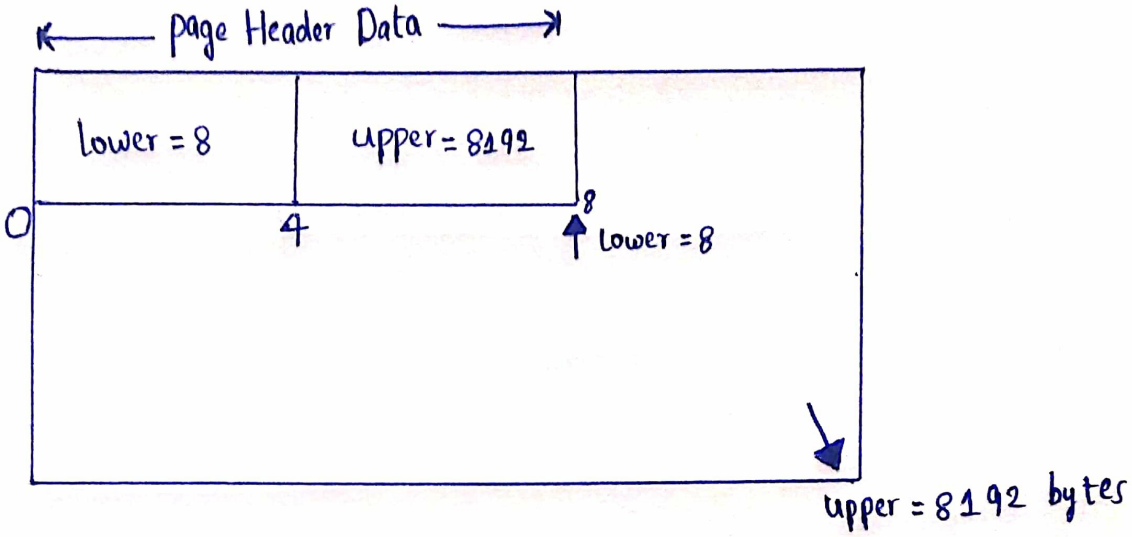
Storage Manager Design Doc - version 0

File / Table Logical Layout




```
pub struct TableHeader {
    pub page_count: u32 // Total pages in table
}
```

Initial page Data



Page = 8 kB

page Header = (lower - 4 bytes + upper - 4 bytes) = 8 bytes

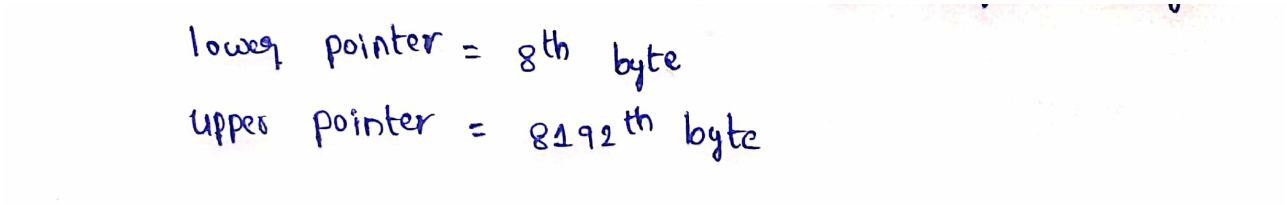


Table - Physical Layout

```
pub struct Table {
    pub data: Vec<u8>, // Fixed-size buffer holds the raw bytes of a table.
}
```

Table - Logical Layout

```
pub const TABLE_HEADER_SIZE: u32 = 8192;

pub struct TableHeader {
    pub page_count: u32, // Total Number of Pages in a Table
}

pub struct Table {
    pub table_header: TableHeader,
    // Pages are laid out consecutively after the table header on disk
}
```

Page - Physical Layout

```
pub const PAGE_SIZE: usize = 8192;

pub struct Page {
    pub data: Vec<u8>  // Fixed-size buffer holds the raw bytes of a page (PAGE_SIZE = 8KB)
}
```

Page Header

```
pub const PAGE_HEADER_SIZE: u32 = 8; // Page Header Size - 8 bytes (4 for lower, 4 for upper)

pub struct PageHeader {
    pub lower: u32,   // Offset to start of free space - 4 bytes
    pub upper: u32,   // Offset to end of free space - 4 bytes
}
```

Item/Tuple Details

```
pub const ITEM_ID_SIZE: u32  = 8;

pub struct ItemId {
    pub offset: u32, // Offset of the item/tuple
    pub length: u32, // Length of the item/tuple
}
```

Logical Page Layout

```
pub struct Page {
    pub header: PageHeader,
    pub item_id_data: Vec<ItemId>,
    // Tuples and their metadata are organized within the page after the header
}
```

- **Table** and **File** are used interchangeably in the document. Both Represent same.

Currently Implemented API's

1. Init Table
2. Init Page
3. Page Count
4. Create Page
5. Read Page
6. Write Page
7. Page Free Space
8. Page Add Data

Ongoing API's

1. Create Table
2. Add Tuple

---

0. init_table API

Description:

- Initializes the **Table Header** by writing the **first page** (8192 bytes) into the table file with 0's. The first 4 bytes represent the **Page Count** (0),

Function:

```
pub fn init_table(file: &mut File)
```

Input:
file: File pointer to update Table Header.

Output:
Table header (first page) initialized with page_count = 0 in the first 4 bytes and remaining bytes set to zero.

Implementation:

1. Move the file cursor to the beginning of the file.
2. Allocate a buffer of 8192 bytes (**TABLE_HEADER_SIZE**) initialized to zero.
3. Write the entire 8192-byte buffer (including the page count) to disk, marking the creation of the first table page.

Test Case:

1. Created a new file to simulate a fresh table.
2. Initialized the table header using **init_table** API, setting the page count to 0.
3. Verified that the file size obtained from **metadata().len()** equals **TABLE_HEADER_SIZE** (8192 bytes), confirming that the entire header page was written successfully.

---

1. init_page API

Description:

- Initializes the **Page Header** with two offset values for **In Memory Page**:
    - **Lower Offset** (PAGE_HEADER_SIZE) → bytes 0..4
    - **Upper Offset** (PAGE_SIZE) → bytes 4..8

Function:

```
pub fn init_page(page:&mut Page)
```

Input:
page: **In Memory Page** to set Header - Lower and Upper Offsets.

Output:
Page header updated with lower and upper offsets.

Implementation:

1. Write the lower offset (PAGE_HEADER_SIZE) into the first 4 bytes of the page header (0..4).
2. Write the upper offset (PAGE_SIZE) into the next 4 bytes of the page header (4..8).

Test Case:

1. Created a new in-memory page with zeroed data.
2. Initialized the page header using init_page API, setting the lower and upper offsets.
3. Checked whether the first 4 bytes were **PAGE_HEADER_SIZE** and the next 4 bytes were **PAGE_SIZE**.

2.page_count API

Description:
To get total number of pages in a file

Function:

```
pub fn page_count(file: &mut File)
```

Input:
file: file to calculate number of pages.

Output:
Total number of pages present in the file.

Implementation:

1. Use the **read_page()** function to read the first page (page ID 0) from the file into memory.

2. Extract the **first 4 bytes** from the in-memory page buffer — these bytes represent the page count stored in the table header.
3. Return the first 4 bytes as page count.

Test Case:

1. Create a temp table file.
2. Initialize it using init_table().
3. Call page_count() to verify it correctly reads 0.

---

3. create_page API

Description:
Create a page in disk for a file.

Function:

```
pub fn create_page(file: &mut File)
```

Input:
file: file to create to a file

Output:

1. Create a page at the end of the file.
2. Update the File Header with **Page Count**.

Implementation:

1. Initializes a new page **in memory** using **init_page** API (update page header - lower and upper).
2. Reads the **current page count** from the file using the **page_count** API.
3. Moves the file cursor to the end of the file.
4. Writes the initialized in-memory page to the file and **updates the file header** by incrementing the page count stored in the first 4 bytes.

Test Case:

1. Verified using File Size, Page Count and Page Headers before and after creating the page using file metadata.

---

4. read_page API

Description:
Reads a page from a disk/file into memory.

Function:

```
pub fn read_page(file: &mut File, page: &mut Page, page_num: u32)
```

Input:
file: file to read from,
page: memory page to fill,
page_num: page number to read

Output:
Populates the given memory page with data read from the file.

Implementation:

1. Calculates the **offset** as **(page_num * PAGE_SIZE)** and moves the file cursor to the correct position.
2. Reads data from that offset position up to **offset + PAGE_SIZE** and copies it into the page memory.

Cases Handled:

1. Checks the file size and returns an error if the requested page does not exist in the file.

Test Case:

- Verified **read_page** API correctly reads one full page — file size equals PAGE_SIZE and data matches the original written content.

5.write_page API

Description:
Write a page from memory to disk/file.

Function:

```
pub fn write_page(file: &mut File, page: &mut Page, page_num: u32)
```

Input:
file: file to write,
page: memory page to copy from,
page_num: page number to write

Output:
Writes the contents of the given memory page to the file at the specified page offset.

Implementation:

1. Calculates the **offset** as page_num * PAGE_SIZE and moves the file cursor to the correct position.
2. copy the contents of the given memory page from offset to offset + PAGE_SIZE positions to the file.

Test Case:

- Verified **write_page** API correctly writes one full page — by using write_page wrote some content to the page and read back the same page to confirm that the written data matches exactly, and the file size is at least PAGE_SIZE.

---

6. page_free_space API

Description:
To calculate the total amount of free space left in the page.

Function:

```
pub fn page_free_space(page: &Page)
```

Input:
page: page to calculate the free space.

Output:
Total amount of freespace left in the page.

Implementation:

1. Read the lower pointer from the first 4 bytes of the page.
2. Read the upper pointer from the next 4 bytes of the page.
3. Calculate free space = upper - lower.
4. Return the free space.

Test Case:
Verified that **page_free_space()** correctly calculates free space (upper - lower) for a newly created data page initialized using init_table() and create_page().
Confirmed that the page header offsets (lower = PAGE_HEADER_SIZE, upper = PAGE_SIZE) and total page count are consistent and accurate.

---

7. page_add_data API

Description:
Adds raw data to the file.

Function:

```
pub fn page_add_data(file: &mut File, data: &[u8])
```

Input:
file: The file to which data should be added.
data: The raw bytes to insert into the page.

Output:
Data inserted in the file.

Implementation:

1. Get the **total number of pages** in the file using page_count API.
2. Read the **last page** into memory using read_page API.
3. Check **free space** in the page using page_free_space API.
4. If the last page has enough free space to store the data and its ItemId
   (i.e., if free_space >= data.size() + ITEM_ID_SIZE):
   a. Calculate the **insertion offset** from the upper pointer.
   start = upper - data.len()
   b. Copy the data bytes into the page buffer starting at this offset.
   c. Update the **upper pointer** in the page header to the new start of free space.
   d. Write the **ItemId entry** (offset and length of the data) at the position indicated by the lower pointer.
   e. Update the **lower pointer** in the page header to account for the newly added ItemId (lower += ITEM_ID_SIZE).
   f. Write the updated page back to disk using write_page API.
5. If the last page does not have enough free space:
   a. [TODO]

- Code - Github (https://github.com/hemanth-sunkireddy/Storage-Manager)
- Code Documentation (https://hemanth-sunkireddy.github.io/Storage-Manager/storage_manager/all.html)
- **Reference 1**: API Formats – Storage Manager Course Assignment Link (http://www.cs.iit.edu/~glavic/cs525/2023-spring/project/assignment-1/)
- **Reference 2**: Postgres Internals – Page Layouts & Data (https://www.postgresql.org/docs/current/storage-page-layout.html)