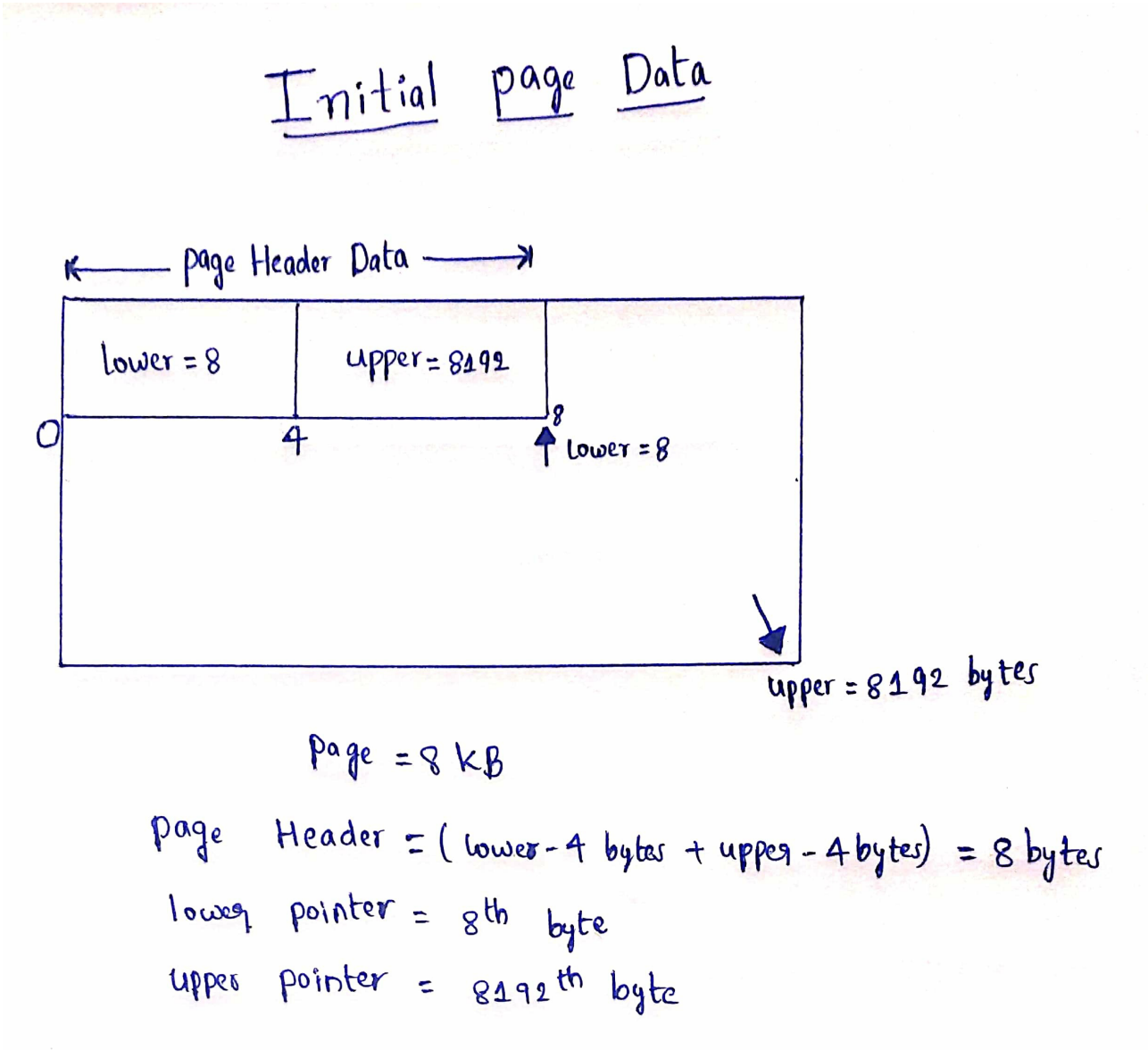
Storage Manager Design Doc - version 0

Page - Physical Layout

```rust
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

```rust
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>,
}
```

Currently Implemented API's

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

---

0. 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:
Create a page at the end of the file.

Implementation:

1. Initializes a new page in memory with all zeros (PAGE_SIZE bytes).
2. Moves the file cursor to the end of the file.
3. Writes the entire zero-filled page to the file, effectively creating a new page on disk.

---

1. init_page API

Description:

- Initializes the **Page Header** with two offset values:
    - **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: 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).

---

2. 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.

3.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.

4.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. Get file size in bytes using file.metadata.len().
2. Divide by PAGE_SIZE to get total pages.
3. Return the page count.

---

5. 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.

---

6. 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)
- **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)