Storage Manager Design Doc - version 0

Page - Physical Layout

```
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
}
```

```
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>,
}
```

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. 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.

2.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.

3.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.

---

4. 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.

---

5. 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:
   a. Calculate the insertion offset from the upper pointer.
   b. Copy the data into the page buffer.
   c. Update the upper pointer in the page header.
   d. Write the page back to disk.
5. If the data does not fit, a new page must be created to insert the data (currently a TODO).