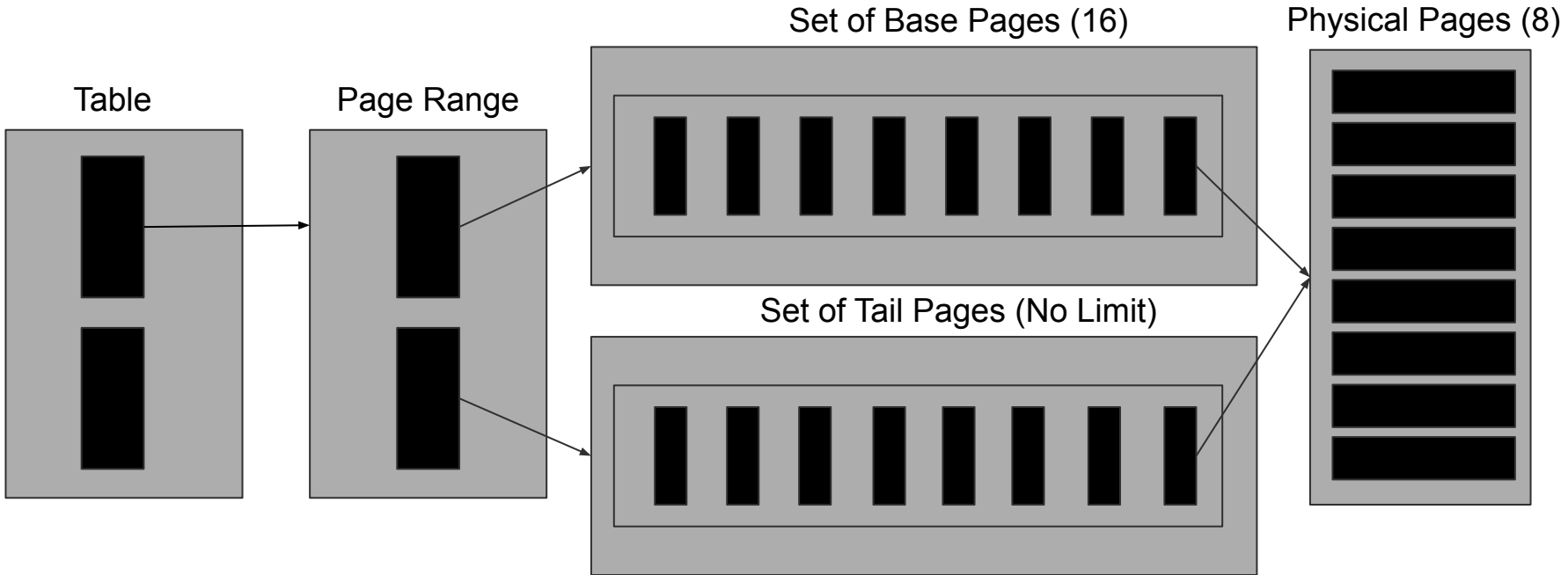
ECS 165A Milestone 1

6 arrays deep

Data Model

Created a hierarchy that has tables -> page ranges -> conceptual pages(base pages & tail pages) -> physical pages(contains physical records)



Database - Create and Drop

```
"""
# Creates a new table
:param name: string         #Table name
:param num_columns: int     #Number of Columns: all columns are integer
:param key: int             #Index of table key in columns
"""
def create_table(self, name, num_columns, key):
    table = Table(name, num_columns, key)
    self.tables.append(table)
    return table
```

```
"""
# Deletes the specified table
"""
def drop_table(self, name):
    for i, table in enumerate(self.tables):
        if table.name == name:
            self.tables.pop(i)
            return
```

Table

- Basic information of the table
- Page directory -> Holds all page Ranges
- Key_dict -> holds locations based on key value
- RID count
  - Tracks next available RID

```
class Table:

    """
    :param name: string         #Table name
    :param num_columns: int     #Number of Columns: all columns are integer
    :param key: int             #Index of table key in columns
    """
    def __init__(self, name, num_columns, key):
        # Want a page per num_columns
        self.name = name
        self.key = key
        # Key is an RID
        self.num_columns = num_columns
        # Page_directory stores the basepages and to find the base page you want
        self.page_directory = []
        self.key_dict = {}
        # Given RID, return a page based off of the RID
        self.index = Index(self)
        self.RID_count = 0
        self.init_key__ = 0
```

Page Range

- Our Page Range contains an array of two arrays:

  [[ ] , [ ]]

  - Base Page array (max 16 base pages)
  - Associated Tail Page array (no limit)
- Local tail record RID counter
  - Tracks Tail RID's for each Page Range
  - Necessary due to RID based location tracking
- Methods:
  - Check if 16 base pages
  - Append base page
  - Append tail page

```python
class PageRange:

    def __init__(self):
        # 1st index is for base pages
        # 2nd index is for tail pages
        self.range = [[],[]]
        self.num_base_pages = 0
        self.num_tail_pages = 0
        self.tail_RID = 0

    def return_page(self):
        pass

    def full(self):
        return self.num_base_pages >= 16

    def append_base_page(self,conceptual_page):
        self.num_base_pages += 1
        self.range[0].append(conceptual_page)

    def append_tail_page(self,conceptual_page):
        # map our base pages to tail pages
        self.num_tail_pages += 1
        self.range[1].append(conceptual_page)

    def merge(self,pages):
        pass
```

Conceptual Page

Indistinguishable Base / Tail Page

- Number of records
- Array of pages (physical pages)
- Function for adding columns
- Full check

```
class ConceptualPage:

    def __init__(self, columns):
        self.pages = []
        self.num_records = 0
        self.add_columns(columns)

    def update_num_records(self,page):
        self.num_records += 1

    def full(self):
        if self.num_records >= 4096:
            return True
        # Check if Full, so if bytes go over 4096 or 4 KB
        return False

    def add_columns(self, columns):
        self.pages.append({}) # Indirection column
        for i in range(1, len(columns) + 4):
            self.pages.append([Page()])
        self.pages[3] = [np.zeros(len(columns))] # Schema Enc column

    def get_page_num(self):
        return self.num_records % 4096 // 512

    def update_RID(self):
        pass
```

Physical Page

- Helper function to convert our values into bytes for our bytearray
- Function to write values into our pages
  - Goes to the latest unwritten space and writes into it.
- Function to retrieve a record based on the record number given

```
def int_to_bytes(val, num_bytes):
    return [(val & (0xff << pos*8)) >> pos*8 for pos in reversed(range(num_bytes))]

class Page:

    def __init__(self):
        self.num_records = 0
        self.data = bytearray(4096)

    def full(self):
        if self.num_records >= 512:
            return True
        # Check if Full, so if bytes go over 4096 or 4 KB
        return False

    def write(self, value):
        offset = self.num_records * 8
        for i in int_to_bytes(value, 8):
            self.data[offset] = i
            offset = offset + 1
        self.num_records += 1
        return True

    def retrieve(self, record_num):
        offset = record_num * 8
        temp = [0,0,0,0,0,0,0,0]
        for i in range(0,8):
            temp[i] = self.data[offset]
            offset  = offset+1
        return int.from_bytes(temp,byteorder='big')
```

Bufferpool Management

- Maintain a **key directory** that maps keys to pages in memory
- Everytime we insert a record, we map that record's key to a physical page
- Ex) Key_dict {key1 : (page_range, base_page, physical_page, index)}
- This allows for fast retrieval of records (average O(1) retrieval w/ Python dict)

Query, Insert

- Find latest Page range, base page, page and record indices by using RID_count
- Check if there is an existing page range with space for our record. If not, make one. Repeat for base page and physical page.
- Add the values and metadata information into the columns of our page
- Map the key of inserted record to its location
- Update our RID and number of records counters

```
def insert(self, *columns):
    new_page_range   = self.table.RID_count % MAX_PAGE_RANGE_SIZE == 0
    page_range_index = self.table.RID_count // MAX_PAGE_RANGE_SIZE
    new_base_page    = self.table.RID_count % MAX_BASE_PAGE_SIZE == 0
    base_page_index  = (self.table.RID_count % MAX_PAGE_RANGE_SIZE) // MAX_BASE_PAGE_SIZE
    new_page         = self.table.RID_count % MAX_PHYS_PAGE_SIZE == 0
    page_index       = (self.table.RID_count % MAX_BASE_PAGE_SIZE) // MAX_PHYS_PAGE_SIZE
    record_index     = self.table.RID_count % MAX_PHYS_PAGE_SIZE

    new_base  = ConceptualPage(columns)
    new_range = PageRange()

    if new_page_range:
        # create new page range
        new_range.append_base_page(new_base)
        self.table.page_directory.append(new_range)
    else:
        if new_base_page:
            self.table.page_directory[page_range_index].append_base_page(new_base)
        else:
            if new_page:
                # append new page to current conceptualpage
                new_base = self.table.page_directory[page_range_index].range[0][base_page_index]
                for i in range(len(new_base.pages)):
                    if not i == 0 and not i == 3:
                        new_base.pages[i].append(Page())
            else:
                new_base = self.table.page_directory[page_range_index].range[0][base_page_index]

    for i, col in enumerate(columns):
        new_base.pages[i+4][page_index].write(col)

    # Get current time value
    current_time = datetime.now().time()
    time_val     = ""
    hour = 0
    # Extract hour * 60, then add to minutes to get total current time in minutes
    for digit in current_time.strftime("%H:%M"):
        if not digit == ":":
            time_val = time_val + digit
        else:
            hour = int(time_val) * 60
            time_val = ""

    time_val = int(time_val) + hour
    values = [self.table.RID_count, time_val]
    self.add_meta(new_base, page_index, values)

    key      = columns[0]
    location = (page_range_index, base_page_index, page_index, record_index)
    self.table.key_dict[key] = location
    self.table.RID_count += 1
    new_base.num_records += 1
    return True
```

Query, Select

- Find location of key from key_dict
- Extract all columns for that record in its base page and store in record rec
- Check if values have been updated in schema encoding column of that record
- If tail record value at that column is not MAX_INT, update record by going to the tail rid pointed by indirection and updating values
- Return the record according to query_columns

```
def select(self, key, column, query_columns):
    location = self.table.key_dict[key] # Assume all keys have been inserted

    p_range, base_pg, page, record = location
    base_pages = self.table.page_directory[p_range].range[0][base_pg].pages
    rid      = base_pages[1][page].retrieve(record)
    indirection = base_pages[0]

    all_columns = []
    # populate with base page values
    for i in range(len(query_columns)):
        all_columns.append(base_pages[i+4][page].retrieve(record))

    # Grab updated values in tail page
    if rid in indirection.keys():
        tail_rid     = indirection[rid]
        tail_page_i  = (tail_rid % MAX_PAGE_RANGE_SIZE) // MAX_BASE_PAGE_SIZE
        page_i       = (tail_rid % MAX_BASE_PAGE_SIZE) // MAX_PHYS_PAGE_SIZE
        tail_page    = self.table.page_directory[p_range].range[1][tail_page_i].pages
        for i, col in enumerate(tail_page[4:]):
            value = col[page_i].retrieve(tail_rid % MAX_PHYS_PAGE_SIZE)
            if value != MAX_INT:
                all_columns[i] = col[page_i].retrieve(tail_rid % MAX_PHYS_PAGE_SIZE)

    columns = []
    for i, col in enumerate(query_columns):
        if col:
            columns.append(all_columns[i])

    key = base_pages[4][page].retrieve(record)
    rec = Record(rid, key, columns)
    return [rec]
```

Query, Update

Get the location of the record based on the key.

Create a query columns list to know which column indices we are updating.

Check to see if the record has been updated (if it is in indirection dict).

If it does have previous updates, then we extract the updates so that we have an updated new tail record.

If it does not have previous updates, then we simply add the passed in value in “columns” to the new tail record.

We then point the base page to the new tail record in the indirection column.

From here, we add an empty new tail page to the latest tail Conceptual Page, and if that is full then we create a new Conceptual Page.

Once the tail page is in the correct location, we then write to the physical pages with the most updated values that have been updated and point the new tail page to the previous tail page in its indirection column.

Lastly, we update the schema encoding for the column(s) that has just been updated.

```
def update(self, key, *columns):
    location = self.table.key_dict[key]# Assume all keys have been inserted

    query_columns = []
    for i, col in enumerate(columns):
        if col != None:
            query_columns.append(1)
        else:
            query_columns.append(0)

    p_range_loc, b_page_loc, page_loc, record_loc = location

    base_page = self.table.page_directory[p_range_loc].range[0][b_page_loc].pages

    indirection = base_page[0]
    page_ind = 512*page_loc + record_loc
    base_schema = base_page[3][page_ind]
    record_rid = base_page[1][page_loc].retrieve(record_loc)
    cols = []

    tail_RID = self.table.tail_RID
    prev_tail_RID = tail_RID
    # Base Page stuff
    if record_rid in indirection.keys():# if update has happened already (ie tail page exists for record)
        tail = indirection[record_rid]
        tail_page_i = tail // 4096
        page_i = (tail % 4096) // 512
        record_i = tail % 512
        # Add updated values to cols
        for i, col in enumerate(self.table.page_directory[p_range_loc].range[1][tail_page_i].pages[4:]):
            if columns[i]==None and not base_schema[i]:
                cols.append(MAX_INT)
            elif columns[i] == None:
                cols.append(col[page_i].retrieve(record_i))
            else:
                cols.append(columns[i])

        prev_tail_RID = tail
    else:# no updates for that record yet
        for col in columns:
            if col != None:
                cols.append(col)
            else:
                cols.append(MAX_INT)

    indirection[record_rid] = tail_RID
    self.table.tail_RID += 1
```

```
## FIGURING OUT WHICH TAIL PAGE TO APPEND TO, CREATE IF DOESNT EXIST
tail_pages = self.table.page_directory[p_range_loc].range[1]
p_range = self.table.page_directory[p_range_loc]
if not tail_pages:# If no tail pages created, create new
    p_range.append_tail_page(ConceptualPage(columns))
elif tail_pages[-1].full():# if tail page full, create new
    p_range.append_tail_page(ConceptualPage(columns))
else:
    tail_pages[-1].pages[3].append(np.zeros(len(columns)))
# Append to most recent tail page
# If the last tail page is full
if tail_pages[-1].num_records % 512 == 0:
    for i, col in enumerate(tail_pages[-1].pages):
        # Not indirection & schema
        if not i == 0 and not i == 3:
            col.append(Page())
tail_page_i = tail_pages[-1].num_records // 512
tail_pages[-1].pages[4][tail_page_i].write(key)
tail_pages[-1].num_records += 1
# write column values into new tail page record
for i, col in enumerate(tail_pages[-1].pages[5:]):
    col[tail_page_i].write(cols[i+1])
# Update Indirection for tail page
tail_indirection = tail_pages[-1].pages[0]
tail_indirection[tail_RID] = prev_tail_RID

tail_schema = tail_pages[-1].pages[3][-1]# Get most recently added schema encoding
# update base page schema encoding
for i, col in enumerate(columns):
    if col != None:
        base_schema[i] = 1
        tail_schema[i] = 1

return True
```

Query, Sum

Iterate through every key in our key dictionary to get the location of each record.

Use the record location to check if the rid is within the range of the start and end index.

If the rid is within the range, check to see if any of the columns have been updated in the schema encoding column.

For the updated columns, retrieve the updated value from the tail page through the indirection column.

For the columns that have not been updated, just grab the value from the base page.

```
"""
def sum(self, start_range, end_range, aggregate_column_index):
    ind = Index(self.table)
    values = ind.locate_range(start_range, end_range, aggregate_column_index)
    return sum(values)
```

Query, Delete

Get the location of record with the given key.

Check the indirection column of that base page to see if that record has been updated.

To identify if a record is to be removed when we merge, the record would have a value in the indirection column indicating it has been updated, but its schema encoding column would be an array of all 0's.

If the record has been updated, change the schema encoding to be an array of all 0's, as well as in the indirection column point to a new tail page with None values.

If the record has not been updated, the schema encoding column will already be an array of all 0's so we just need to point to a new tail page with None values in the indirection column

```
def delete(self, key):
    # Grab location of base record
    baseR_loc = self.table.key_dict[key]
    baseR_p_range, baseR_base_pg, baseR_pg, baseR_rec = baseR_loc
    base_pages    = self.table.page_directory[baseR_p_range].range[0][baseR_base_pg].pages
    base_rid      = base_pages[1][baseR_pg].retrieve(baseR_rec)
    base_schema_i = MAX_PHYS_PAGE_SIZE*baseR_pg + baseR_rec
    # Check indirection column to see if has been updated
    indirection = base_pages[0]
    updated     = base_rid in indirection.keys()
    n_cols      = self.table.num_columns
    # If not updated, add tail page with MAX_INT vals and add to indirection
    if not updated:
        # Update to add tail page with None for all values
        self.update(key, *[None]*n_cols)
    else:
        # Change base schema to all 0's, then update which gives None tail page
        base_pages[3][base_schema_i] = np.zeros(n_cols)
        self.update(key, *[None]*n_cols)

    return True
```

Problems & Future Considerations

- Lots of reused code (can be moved into functions)
- Implement Config.py
  - Universal constants(Max_page_size,Max_page_range,Max_concept_page)
  - Helper functions
- Overhaul literally everything
  - Simplify and consolidate code
  - Implement B-Trees for Indexing instead of one large dictionary