L-Store: Milestone 2

ECS 165A: Database Systems

Yiling Chen, Tina Young, Olivia Tobin

3 Main Parts

1

Durability & Bufferpool Extension

2

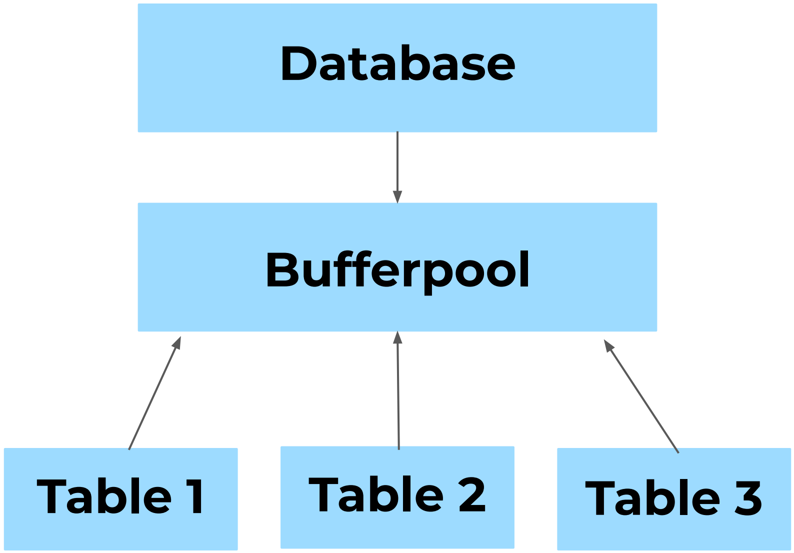
Data Reorg

3

Index

Durability & Bufferpool Extension

Bufferpool Class



Database

Bufferpool

Table 1

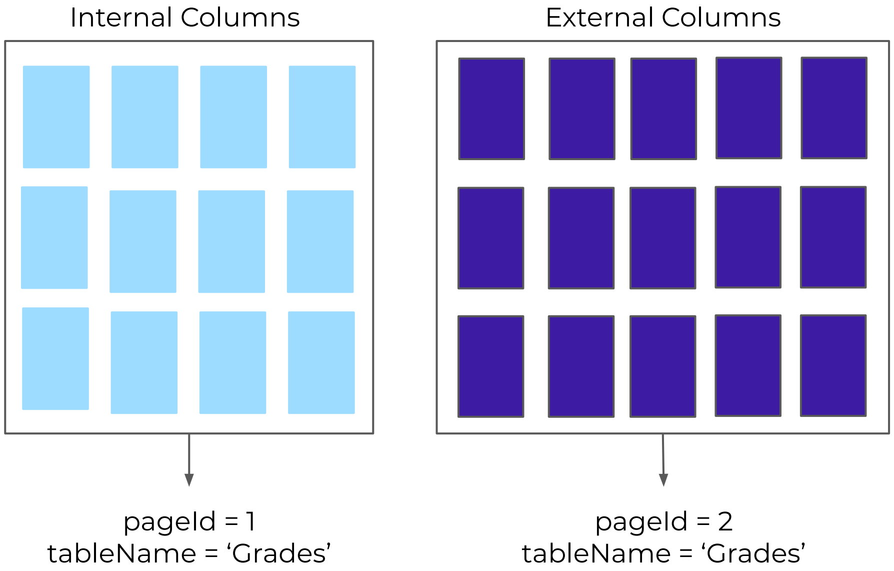
Table 2

Table 3

Data Storage on Disk: Base Page

🟦 Internal = meta records (RID, SE, IND, TIME)

🟪 External = the records



Internal Columns

External Columns

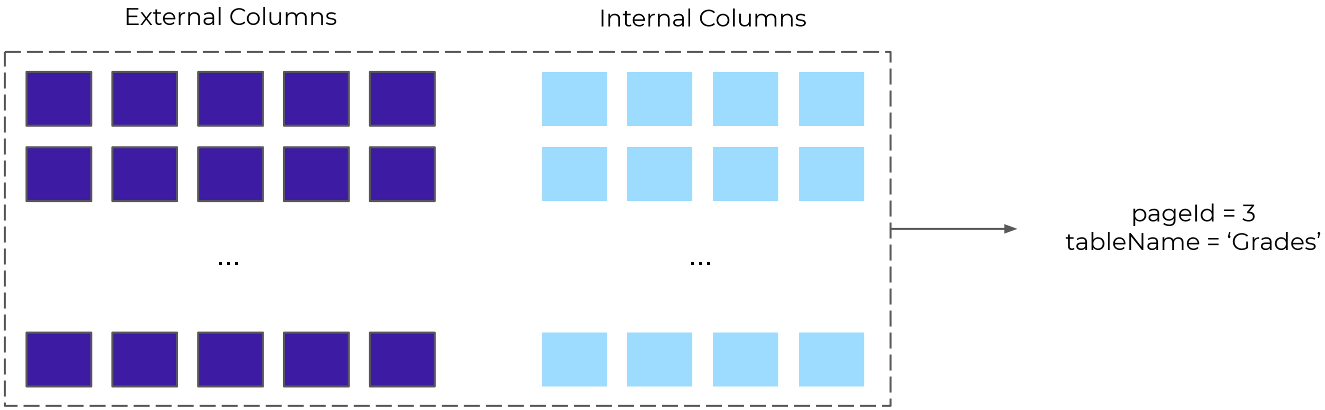
pageId = 1
tableName = ‘Grades’

pageId = 2
tableName = ‘Grades’

Data Storage on Disk: Tail Page

🟪 External = the records

🟦 Internal = meta records (RID, SE, IND, TIME)



External Columns

Internal Columns

...

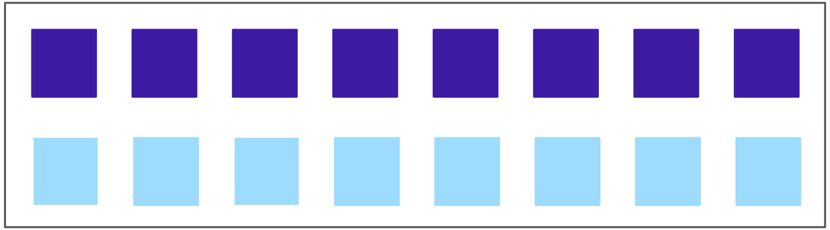
...

pageId = 3
tableName = ‘Grades’

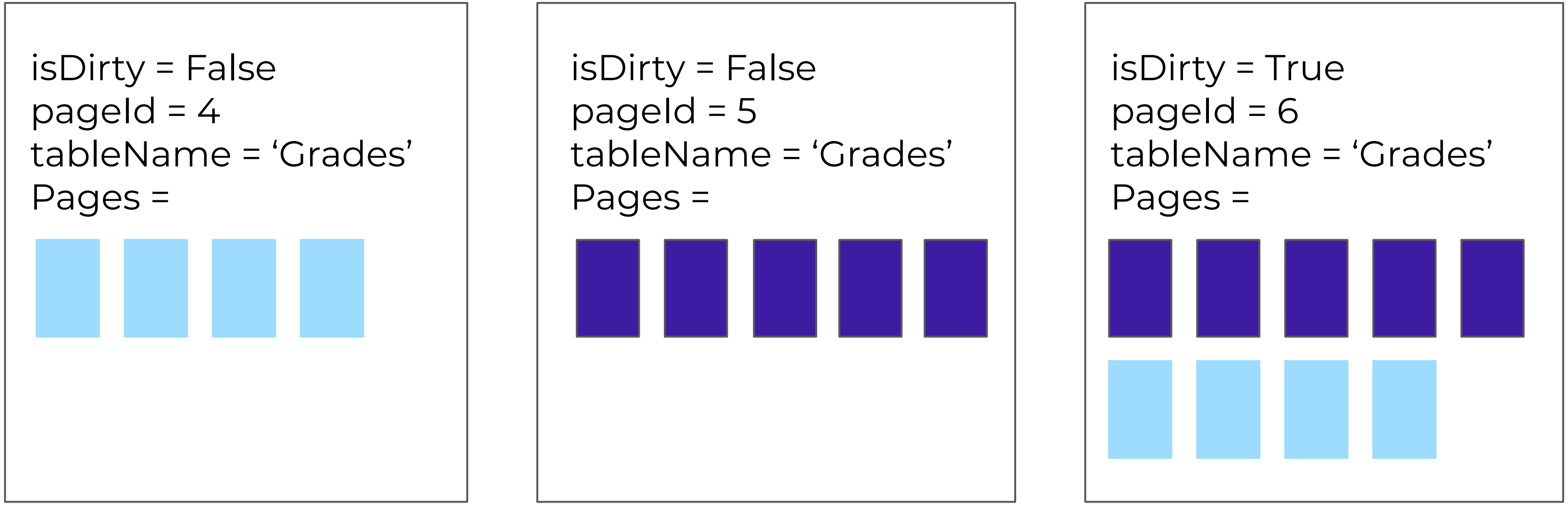
Bufferpool Consists of Slot Objects

Size can be set inside in config file. We set it to 16 for this milestone

🟪 = Dirty 🟦 = Clean



Slot Objects



Slot Object for a Base Page Internal

isDirty = False
pageId = 4
tableName = ‘Grades’
Pages =

Slot Object for a Base Page External

isDirty = False
pageId = 5
tableName = ‘Grades’
Pages =

Slot Object for a Tail Page

isDirty = True
pageId = 6
tableName = ‘Grades’
Pages =

Key Database Functions

```
def open(path)
    # Set the path for the directory that will hold all of the record information (disk)
    self.bufferpool.set_path(path)
    self.path = path
    if not os.path.exists(path):
        os.makedirs(path)

def close()
    # Write all of the slots in the Bufferpool and write individual table information
    self.bufferpool.write_all()
    for i in range(0, len(self.tables)):
        table = self.tables[i]
        table.save_table()
```

Key Bufferpool Functions

def read_file(page_id, table_name, num_columns):

- *Read data from file named [table_name]_[page_id].txt*
- *Put it in a format that the functions in query can understand*
    - *Slot Object*
- *Put slot object into Bufferpool*

def write_file(slot_object):

- *Write the specified Bufferpool Slot to a file*
- *First 8 bytes are always the number of records*
- *Second element is always the lineage of each page*
- *Then, the byte arrays of the pages*

Pinning Pages: Move_to_Front()

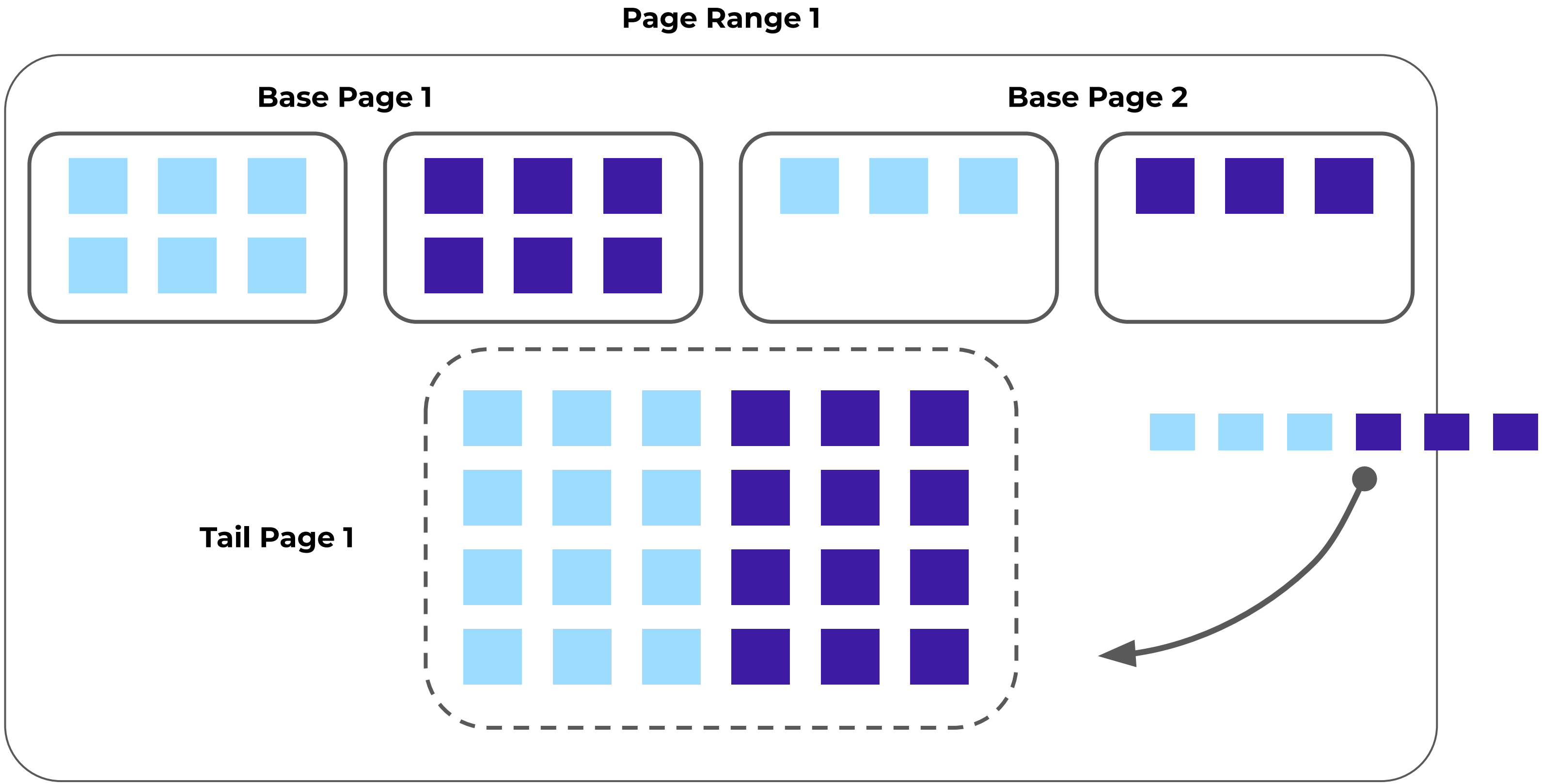
- Pop slot object and move it to index 0 to indicate that it has recently been accessed
- When a page needs to be evicted, automatically evict the last slot so that you're accessing what hasn't been accessed recently

Dirty Page

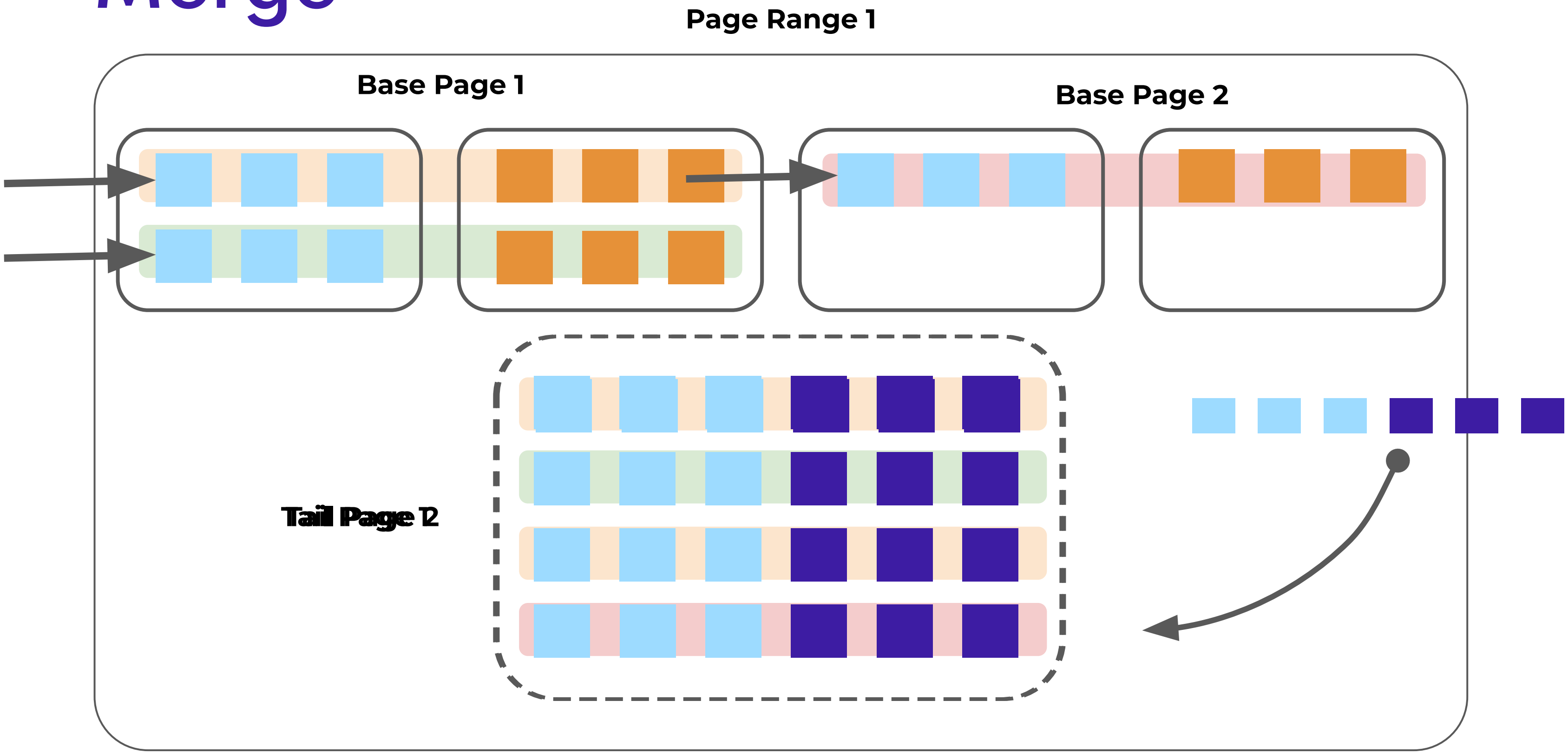
- Whenever the contents of a page have been changed, mark it as dirty

Data Reorg

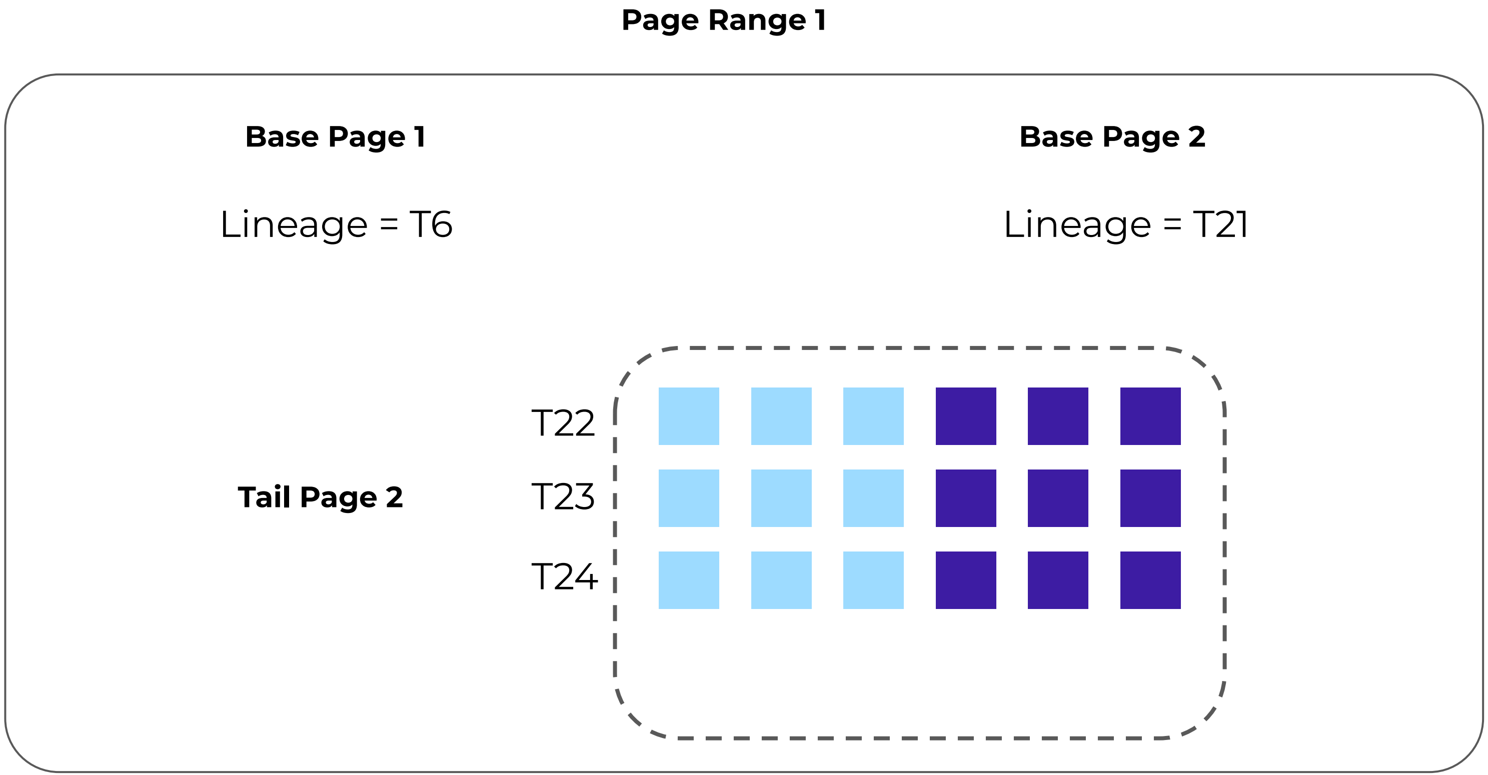
Merge



Merge



TPS = lineage



Page Range 1

Base Page 1

Lineage = T6

Base Page 2

Lineage = T21

Tail Page 2

T22

T23

T24

Index

Page, Primary Key, Index Directory

	Page Directory	Primary Key Directory	Index Directory
Data Structure	Hashmap (Dict)	Hashmap (Dict)	Hashmap (Dict)
Key	RID	PRIMARY KEYS	RECORD VALUE (from specified column)
Value	PageID_INT, PageID_EX, Offsets	RID	RID_LIST

Index Example

GOAL: SELECT(3, 2, [1, 1, 1, 1, 1])

CREATE_INDEX(2)

create_index() and drop_index()

```
# we have initialized a list of empty indices
self.indices = [None] * num_columns

def create_index(column):
     index_dict = {}                                   # initialize a dictionary
     key_list = list(primary_key_directory.keys())     # find all primary keys
     for i in len(key_list):
          rid = primary_key_directory[key[i]]          # find RID for each key
          most_updated = get_most_updated(rid)    # get most updated records
          value = most_updated(column)                 # get column value
          index_dict[value].append(rid)           # put value into dictionary
     self.indices[column] = index_dict

def drop_index(column):
     # remove from the indice lists
     self.indices[column] = None
```

Index Example

GOAL: SELECT(3, 2, [1, 1, 1, 1, 1])

CREATE_INDEX(2)

SELECT(3, 2, [1, 1, 1, 1, 1])[i].columns

[[63725, 14, **3,** 2], [37264, 6, **3**, 16]...]

UPDATE(63725, *([None, None, **5**, None]))

update_index()

This function is called inside **`UPDATE()`**: if the index dictionary for that particular column has already been created, then we will also update the index when updating.

```
def update_index(rid, update_value, original_value, column):
    # this function will only happen if the hashmap for that column exists
    index_dict = self.indices[column]
    rid_list = index_dict[original_value]
    # remove from previous index
    index_dict[original_value].pop(rid_list.index(rid))
    # check for case of DELETE()
    if new != None:
         index_dict[update_value].append(rid)
```

Index Example

GOAL: SELECT(3, 2, [1, 1, 1, 1, 1])	
CREATE_INDEX(2)	
SELECT(3, 2, [1, 1, 1, 1, 1])[i].columns	[[63725, 14, **3,** 2], [37264, 6, **3**, 16]...]
UPDATE(63725, *([None, None, **5**, None]))	
SELECT(3, 2, [1, 1, 1, 1, 1])[i].columns	[[37264, 6, **3**, 16]...]
SELECT(5, 2, [1, 1, 1, 1, 1])[i].columns	[[63725, 14, **5**, 2]...]

locate() and locate_range(): returns RID

```
def locate(value, column)

    # check if the index has been created, if it does, it’s the most updated
    if self.indices[column] == None:
         self.create_index(column)
    index_dict = self.indices[column]
    rid = index_dict[value]
    return rid

def locate_range(start, end, column)

    # use locate(value, column)
    index_dict = self.indices[column]
    for i in range(start, end):
         rid_list.append(self.locate(i, column))
    return rid_list
```

Things to Improve Upon for M3

- Support multithreading
  - Snapshot
- Separate the each column into separate files
  - In order to compress base pages
- Clean up the code
  - Keep logics and most of the operations inside table.py and bufferpool.py
- Experiment with more efficient indexing strategies

Thank You!

Performance Comparison

Performance Comparison for Page Edits