Froyo - Milestone 1
Single-threaded, In-memory L-Store

Developers: Heming Ma, Brian Lai, Roy Yi, Ruyi Yang, Yongxin Xu
Schedule

1. Design and implementation (8 min)
2. Demo (4 min)
3. Q/A (8 min)
Bufferpool

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>_cache: OrderedDict</td>
</tr>
<tr>
<td>_capacity: int</td>
</tr>
<tr>
<td>get_page_range(page_range_id)</td>
</tr>
<tr>
<td>put(page_range_id, page_range)</td>
</tr>
</tbody>
</table>
Index

Index interface

- _num_of_columns: int
- _primary_key_column: int
- _indices: [Index]

- insert(column, key, rid)
- locate(column, value)
- delete(primary_key)
- update(primary_key, columns)
- get_range(begin, end, column)
| name: string |
| _num_of_columns: int |
| _primary_key_column: int |
| _available_record_id: int |
| _index: Index |
| _page_ranges: [PageRange] |
| _bufferpool: Bufferpool |

insert(columns)
select(index_key, column, query_columns)
delete(primary_key)
update(primary_key, columns)
get_sum()
Page Range

base_id (0-19)

0

base_id (20-39)

1
Layout

indirect  sid  age  grade

indirect  col 0  col 1  col 2

base_id: 0

base_id: 1

Offset: 5
Query Operations

**Insert:** insert new entry in last base page

**Delete:** set indirection column to invalid flag

**Update:** insert new entry in last tail page
point base page indirection column to latest tail entry

**Process_rid:** index base entry with rid, go to latest tail entry if previously updated

**Select/sum:** scan through all rids in table to find all matches (using process_rid)
Index

- B+ tree
- Hash Table (Open-addressing)
- Hash Table (Separate Chaining)
- R-Hash Table (Open-addressing)
- R-Hash Table (Separate Chaining)
# HashTable & B-tree Comparison

<table>
<thead>
<tr>
<th></th>
<th>B+ tree</th>
<th>Hash-table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search operation</strong></td>
<td>root-to-leaf ( O(\log n) )</td>
<td>Single I/O per look-up ( O(1) )</td>
</tr>
<tr>
<td></td>
<td>Sequential search available (sorted container)</td>
<td></td>
</tr>
<tr>
<td><strong>Memory management</strong></td>
<td>Easy memory allocation/deallocation (split node / merge node)</td>
<td>Hard to find an algorithm that expand/shrink the table size (rehash / open-address)</td>
</tr>
<tr>
<td><strong>Insertion /deletion</strong></td>
<td>( O(n\log n) ) for batch insertion/deletion Have to maintain self-balance</td>
<td>( O(1) ) for deletion and insertion ( O(n) ) for batch operation</td>
</tr>
</tbody>
</table>
B-tree Implementation

Challenge:
batch insertion takes $O(n \log n)$

Future Improvement:
Split node strategies
Size of b+ tree

(reference: https://www.programiz.com/dsa/b-plus-tree)
B+ tree performance

```
yruy0705@ad3.ucdavis.edu@pc19:/ecs165/test1/ECS165A-Milestone1$ python3 __main__.py
Inserting 10k records took: 0.179706857
Updating 10k records took: 0.271136593
Selecting 10k records took: 0.1296896100000007
Aggregate 10k of 100 record batch took: 0.03496454299999996
Deleting 10k records took: 0.050827964
```
Hash Table Implementation

We employ two types of strategies, separate chaining and open-address, to experiment the efficiency, with built-in Python hash function.

Open-Address (Using Python dictionary):

- No collision, meaning each entry only hold one slot.
- Insertion is more expensive for finding in an empty entry and allocating more space.
- Random probing algorithm.
- Selection in $O(1)$

Separate-Chaining:

- With constant table size and collision.
- Insertion in $O(1)$
- Selection in $O(n/k)$, in the range between $O(1)$ and $O(n/k)$.
R-Hash Table Implementation

R-HashTable, differing from HashTable, provides linklist between each index in a sequential order that optimizes range queries.

Comparing to Hash Table, R-Hash Table:

- Has slower insertion, update, and deletion due to the maintenance of sorted keys, which is $O(n \log n)$.
- Uses seeds to shorten range searches.
- Speeds up range queries in $O(n/k)$, comparing to $O(n)$ by Hash Table, where $k$ is the number of seeds.
- Is Read-optimized and update-Heavy.
Performance for Insertion

![Graph showing time (in seconds) vs. number of records (in 5k intervals) for different data structures: hash_dic, r_hash_dic, and bptree. The graph indicates that as the number of records increases, the time taken for insertion also increases significantly for the bptree, while the other structures show more linear growth.]
Performance for Selection

![Graph showing performance time for selection with different record numbers and methods like hash_dic, r_hash_dic, and bptree.]
Performance for Range Selection

![Graph showing performance comparison for range selection with different methods: hash_dic, r_hash_dic, and bptree. The x-axis represents the number of records (5k), and the y-axis represents time in seconds. The graph illustrates how each method performs under varying loads, with hash_dic showing a higher aggregate time compared to the others, especially notable at higher record counts.]
Performance for Deletion

![Graph showing performance for deletion with different data structures over varying number of records.](image-url)
Performance for Different Indexes
Questions & Demo