

Lecture 5: Slotted Page



Logistics

- Point Solutions App
 - Session ID: **database**
- Programming assignment 1 due on Sep 10 via Gradescope
- One-page introduction sheet due on Sep 10 via Gradescope

Recap

- Smart Pointers
- Smart Field
- Heap vs Stack
- Page class
- Simplifying serialization
- Static method for deserialization

Simplifying Page Serialization

Field Class Serialization

```
void Field::serialize(std::ostream &out) {  
    out << type << ' ' << data_length << ' ' ;  
    if (type == STRING) {  
        out << data.get() << ' ' ;  
    } else if (type == INT) {  
        out << *reinterpret_cast<int *>(data.get()) << ' ' ;  
    } else if (type == FLOAT) {  
        out << *reinterpret_cast<float *>(data.get()) << ' ' ;  
    }  
}
```


Page::deserialize(filename)

Merge Creation + Loading

Direct Tuple Loading

No Temporary Page Object

```
auto loadedPage = Page::deserialize(filename);  
  
// Deserialize from disk  
// Page page2;  
// page2.read(filename);
```


Lecture Overview

- Tuple Deletion
- Slotted Page
- File Management
- Index Construction

Tuple Deletion



Tuple Deletion

Tuple Deletion Process

deleteTuple method

Check vector bounds

Update page size

Remove tuple from vector

```
void Page::deleteTuple(size_t index) {  
    if (index >= tuples.size()) {  
        std::cout << "Tuple index out of range. ";  
        return;  
    }  
    used_size -= tuples[index]->getSize();  
    tuples.erase(tuples.begin() + index);  
}
```


Tuple Deletion

Logic introduced to delete tuples

**Simulate
Data Lifestyle**

**Skip Deletion
Periodically**

**Database
Cleanup**

```
// Skip deleting tuples only once every hundred tuples
if (tuple_insertion_attempt_counter % 100 != 0) {
    page.deleteTuple(0);
}
```


Data Inconsistency

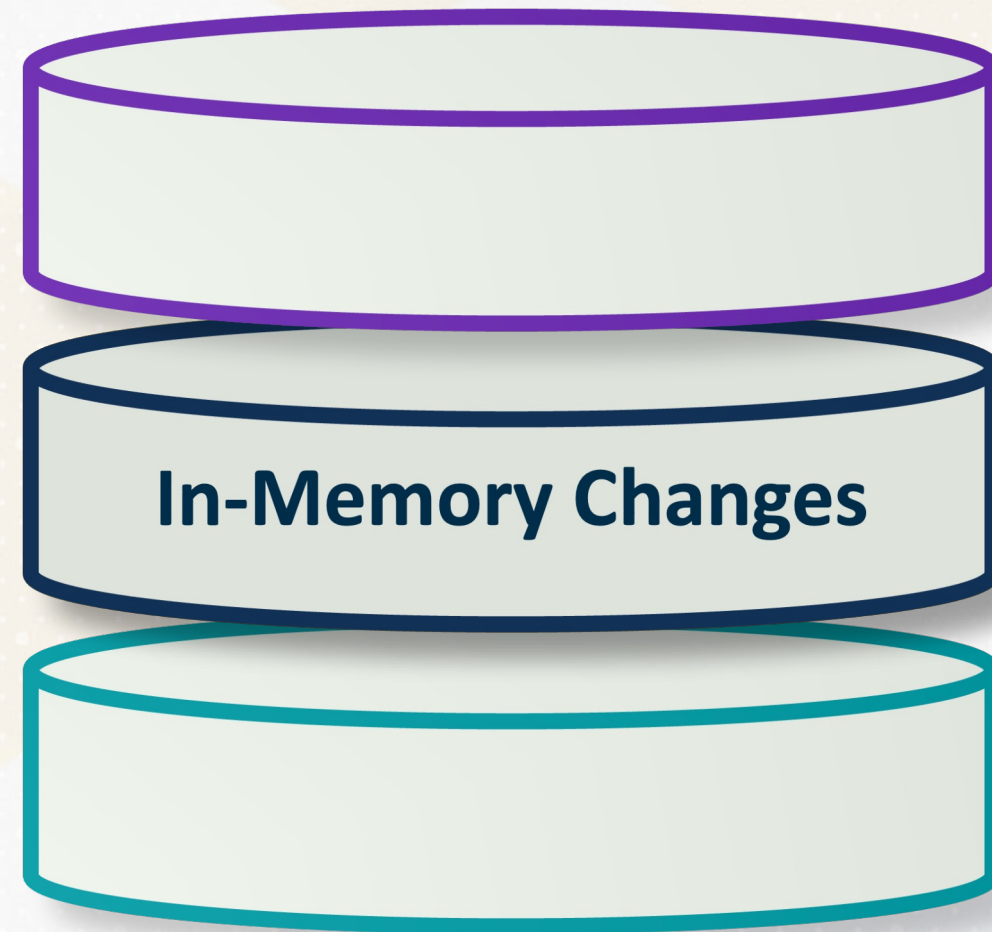
In-Memory Deletion

Changes do not reach disk

loadedPage2 mismatch

```
// Serialize to disk
db.page.write(filename);
// Deserialize from disk
auto loadedPage = Page::deserialize(filename);
// PROBLEM: Deletion only in memory, not on disk
loadedPage->deleteTuple(0);
// Deserialize again from disk -- page unchanged
auto loadedPage2 = Page::deserialize(filename);
```


Data Inconsistency



Data Inconsistency Issues

In-Memory Changes Not Persisted

Memory and Disk State Discrepancy

Data Synchronization

Trigger Serialization Process

Disk State Updated

```
std::cout << "Deleting slots 0 and 7 \n";  
loadedPage->deleteTuple(0);  
loadedPage->deleteTuple(7);  
  
loadedPage->write(filename);
```

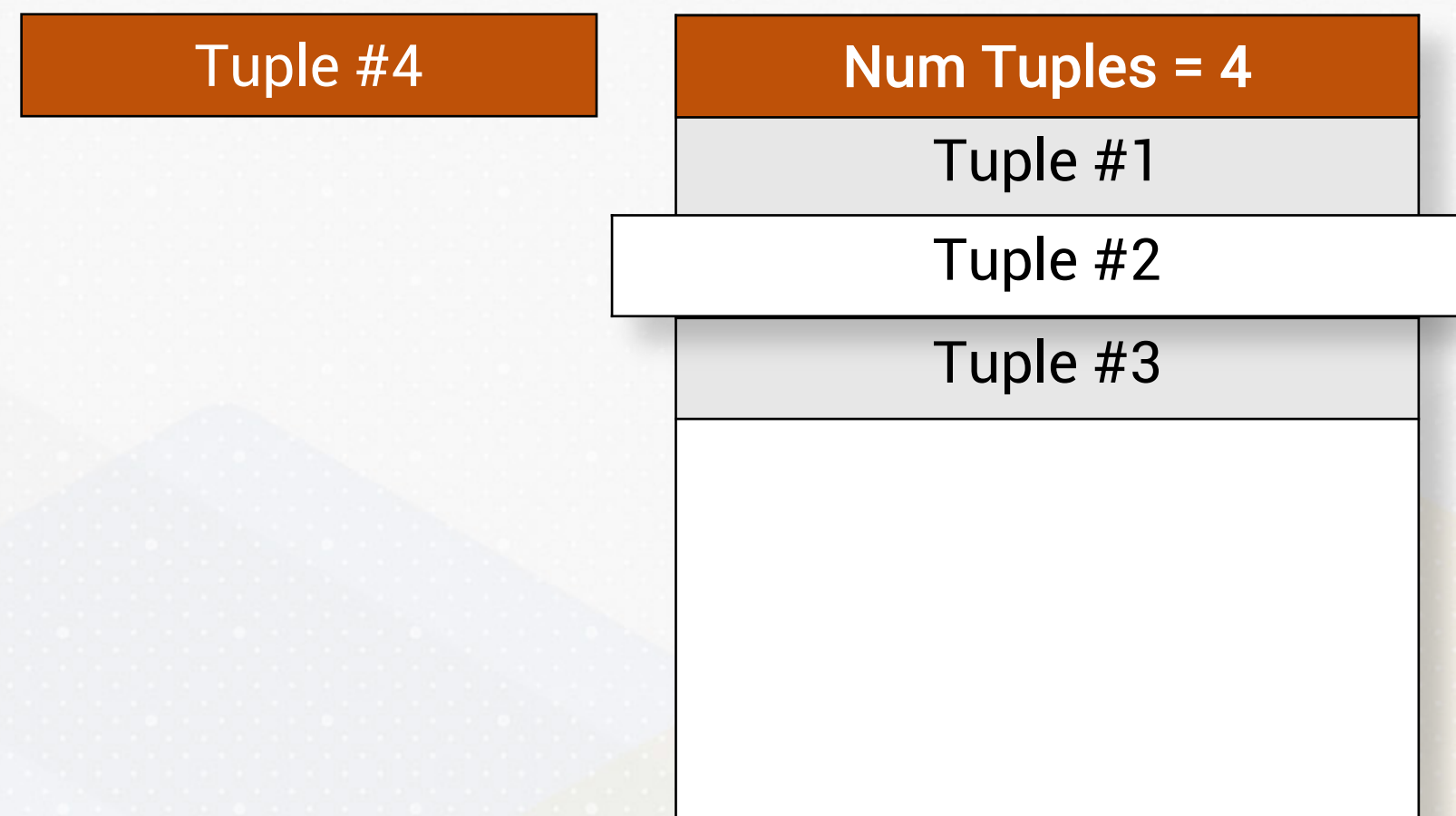

Data Synchronization

Modifications reflected during deserialization

```
// Deserialize again from disk -- page is updated this time  
auto loadedPage2 = SlottedPage::deserialize(filename);  
loadedPage2->print();
```

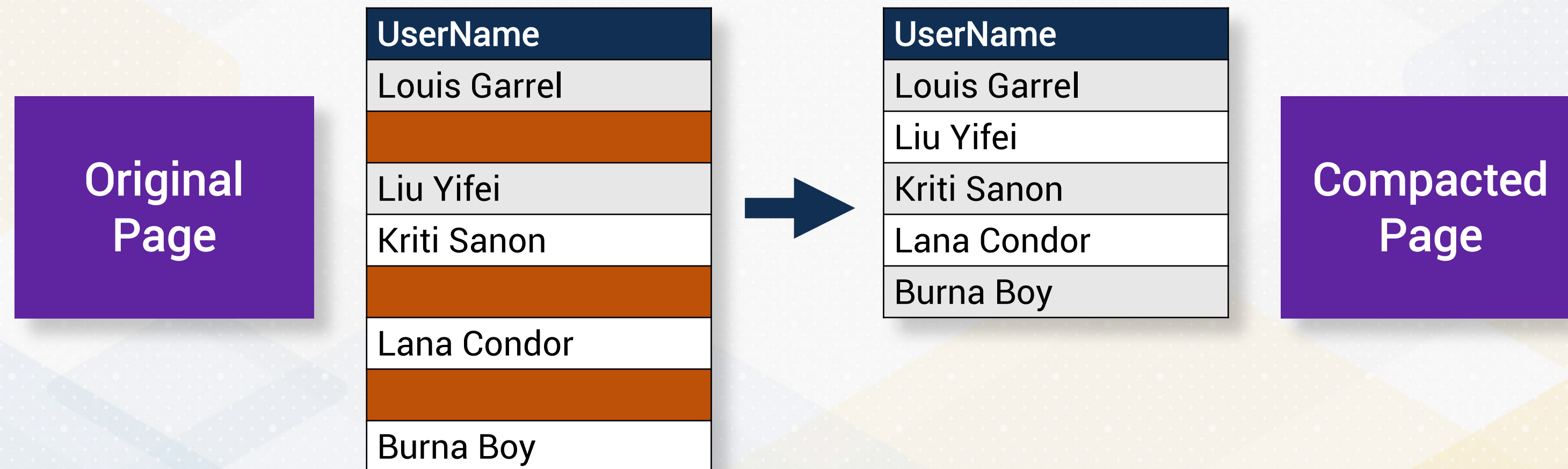

Limitations of Page Class: Linear Scan

- Finding space for a new tuple requires a linear scan.
- Poor handling of variable-length tuples leads to wasted space.



Limitations of Page Class: Compaction

- Periodically, we must "compact" this "fragmented" page, moving tuples around to consolidate free space.



Slotted Page



Slotted Page



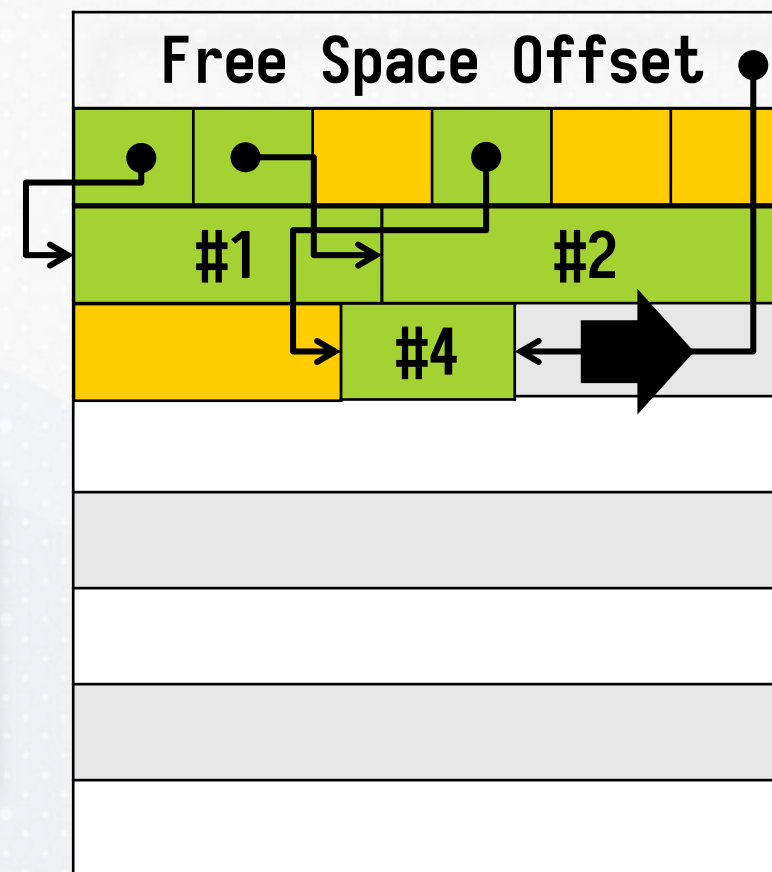
Slots are entries in a page's "header" section that keep track of where the tuples are stored within the page.

```
struct Slot {  
    uint16_t offset; // Offset of the tuple within the page  
    uint16_t length; // Length of the tuple data  
};
```


Slotted Page



Slots are entries in a page's "header" section that keep track of where the tuples are stored within the page.



Slot Array

Variable
Length
Tuples

Slotted Page

```
class SlottedPage {  
    std::unique_ptr<char[]> page_data;  
    std::vector<Slot> slots;  
    size_t free_space_offset;  
    ...  
};
```

Slots enable direct access to any tuple by index.

Anatomy of a Slotted Page



Slotted Page

Tracks:

- Metadata
- Slot Numbers
- Free Space Offset

Slots Mark Tuple
Data Start on Page

```
class SlottedPage {  
    std::unique_ptr<char[]> page_data;  
    std::vector<Slot> slots;  
    size_t free_space_offset;  
    ...  
};
```


Slotted Page: Adding Tuple

```
bool addTuple(std::unique_ptr<Tuple> tuple) {  
    ...  
    size_t slot_itr = 0;  
    Slot *slot_array =  
        reinterpret_cast<Slot *>(page_data.get());  
    for (; slot_itr < MAX_SLOTS; slot_itr++) {  
        if (slot_array[slot_itr].empty &&  
            slot_array[slot_itr].length >= tuple_size) {  
            break;  
        }  
    }  
    // Determine tuple placement and update slot  
    return true;  
}
```

Checks for
Available Space

Identifies Correct
Tuple Slot

Ensures Efficient
Space Utilization

Slotted Page: Deleting Tuple

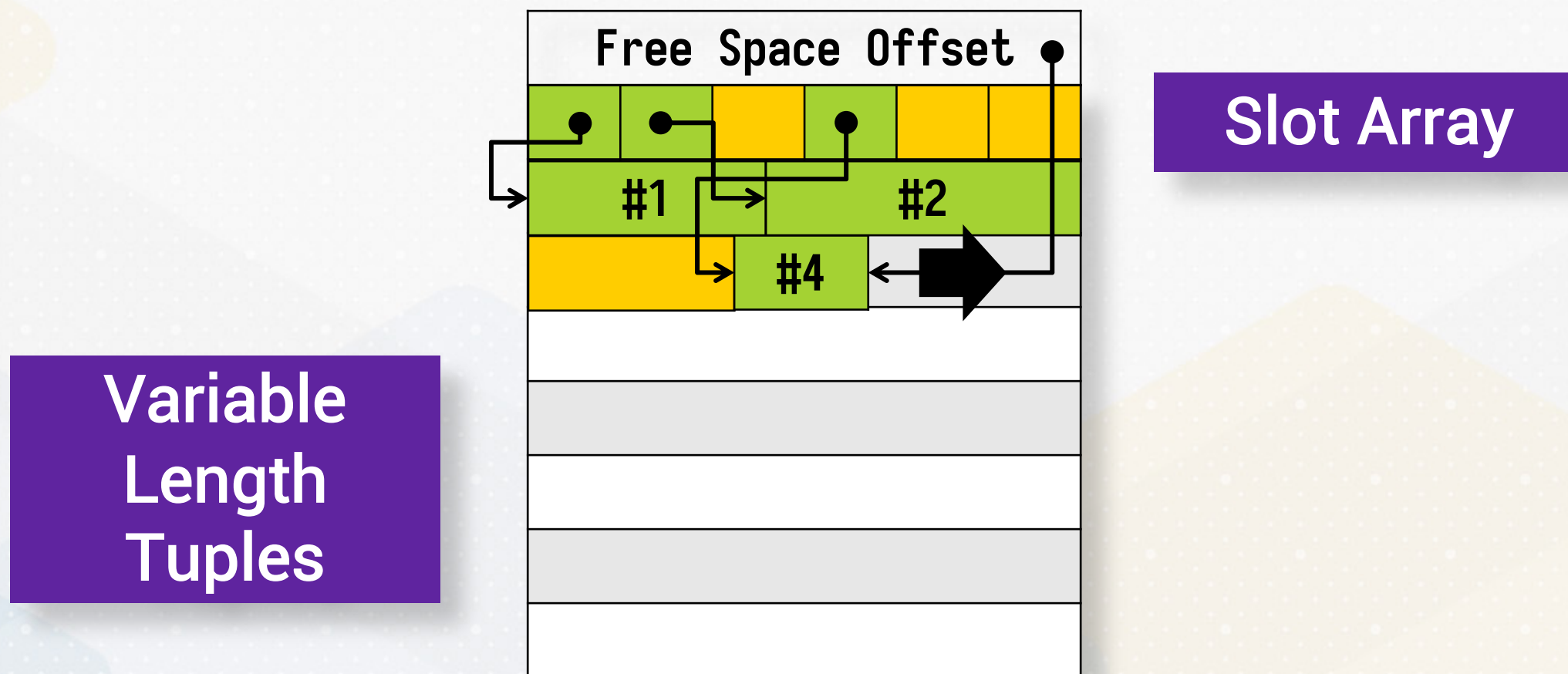
```
void deleteTuple(size_t index) {  
    Slot *slot_array = reinterpret_cast<Slot *>(page_data.get());  
    if (index < MAX_SLOTS && !slot_array[index].empty) {  
        slot_array[index].empty = true;           // Mark the slot as empty  
        used_size -= slot_array[index].length;    // Reclaim space  
    }  
}
```



- Marks a tuple as deleted within the **Slotted Page** by updating the slot array
- Frees space for future insertions or updates
- Efficient Space Reclamation

Slotted Page: Updating Tuple

- **Updates that change tuple size:**
 - we can either update the tuple in-place (if space permits)
 - or move the tuple within the page and update the slot with the new offset and length.



Addressing Limitations of Page Class

```
bool addTuple(const char *tupleData, size_t length) {  
    if (free_space_offset + length > PAGE_SIZE) return false; // Check space  
    page_data[free_space_offset] = ... // Copy tuple data  
    slots.push_back({free_space_offset, length}); // Update slot  
    free_space_offset += length; // Move free space offset  
    return true;  
}
```

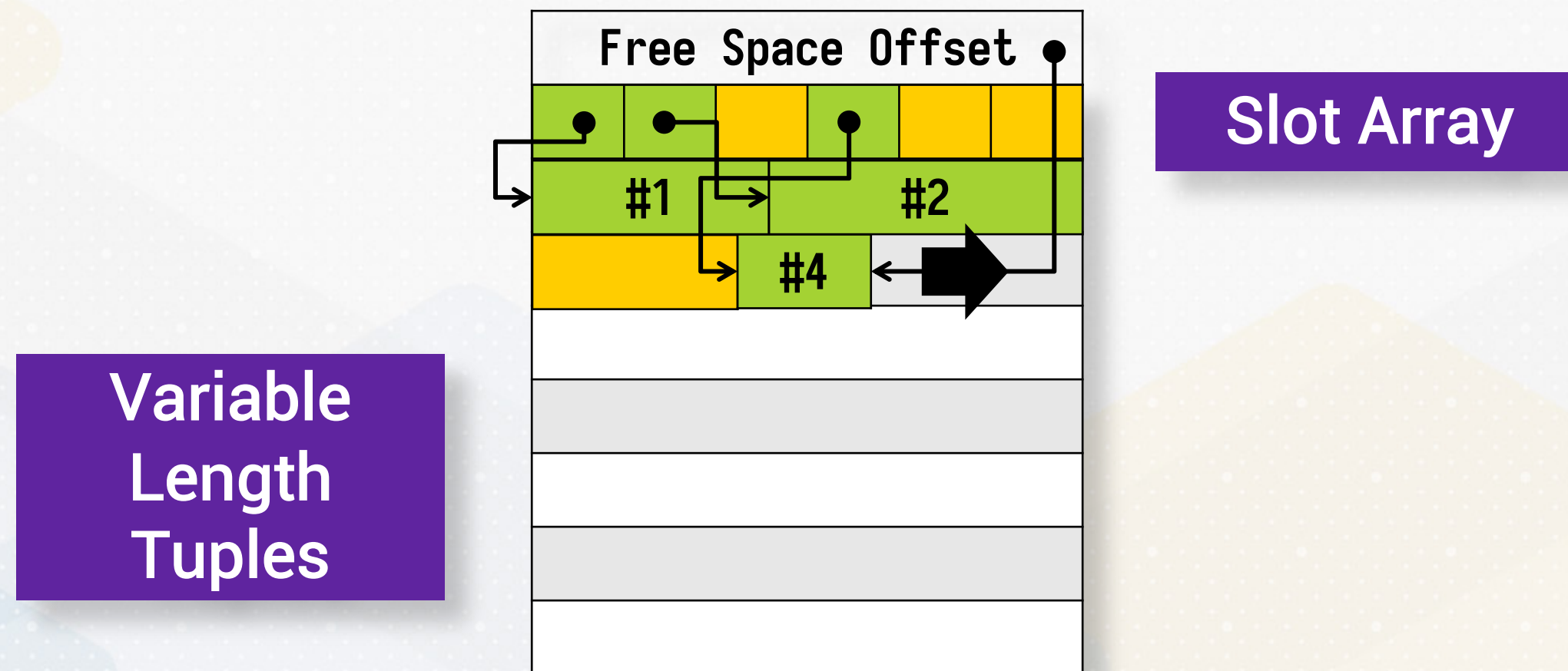


Steps for Adding A Tuple to Almost Full Page:

- Check Page Header for Contiguous Free Space
- Directly Access the Slot for Modification

Addressing Limitations of Page Class

- Each slot points to a variable-length tuple, maximizing space utilization and minimizing internal fragmentation.
- Compaction is simplified. Tuples can be moved to compact the page, and only the slot offsets need to be updated.



File Management



Database File Management

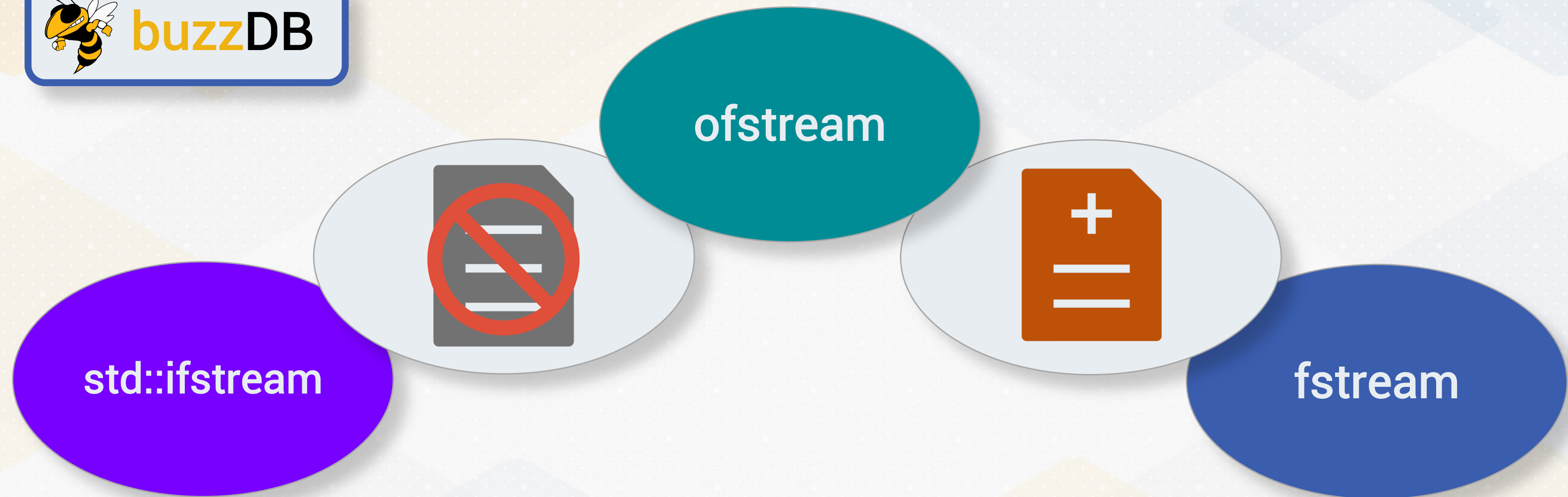


buzzDB

- BuzzDB manages a single Slotted Page instance
- Limits the database to operating with a single page

```
class BuzzDB {  
private:  
    std::fstream file;  
    // a vector of Slotted Pages acting as a table  
    std::vector<std::unique_ptr<SlottedPage>> pages;  
  
public:  
    BuzzDB() { file.open(database_filename, std::ios::in | std::ios::out); }  
  
};
```


Database File Management



```
std::ifstream infile(database_filename);  
if (!infile.good()) {  
    std::ofstream outfile(database_filename);  
}
```


Database File Management



The database file is then opened with read and write permissions

The constructor calculates the number of pages in the database by seeking to the end of the file and dividing the file size by **PAGE_SIZE**

```
file.open(database_filename, std::ios::in | std::ios::out);  
file.seekg(0, std::ios::end);  
num_pages = file.tellg() / PAGE_SIZE;
```



std::fstream

Read

ios::in

Write

ios::out

Extending Database File

If no pages are found, it calls **extendDatabaseFile()** to add an initial empty page, ensuring the database is ready for data insertion.

```
if (num_pages == 0) {
    extendDatabaseFile();
}

void extendDatabaseFile() {
    auto empty_slotted_page = std::make_unique<SlottedPage>();
    file.seekp(0, std::ios::end);
    file.write(empty_slotted_page->page_data.get(), PAGE_SIZE);
    file.flush();
    // Load the new page into memory...
}
```


Extending Database File

The **extendDatabaseFile()** function handles the low-level file operations required to append a new page to the database file.

```
void extendDatabaseFile() {  
    auto empty_slotted_page = std::make_unique<SlottedPage>();  
    // Write the buffer to the file, extending it  
    file.seekp(0, std::ios::end);  
    file.write(empty_slotted_page->page_data.get(), PAGE_SIZE);  
    file.flush();  
    // Update number of pages  
    num_pages += 1;  
}
```


Extending Database File

0

Page #1

1

Page #2

2

Page #3

3

Page #4

4

Page #5

0

4096 B

8192 B

12 KB

16 KB

Inserting Data into Database File



buzzDB

- BuzzDB checks existing pages to insert new tuple
- `extendDatabaseFile` adds new page if all are full

```
bool status = try_to_insert(key, value);  
// Try again after extending the database file  
if (status == false) {  
    extendDatabaseFile();  
    bool status2 = try_to_insert(key, value);  
    assert(status2 == true);  
}
```


Loading Pages from Database File

Loading involves
iterating through the
existing pages

Each page
deserialized into a
SlottedPage object

Each **SlottedPage**
object added to the
pages vector

```
for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {  
    std::unique_ptr<SlottedPage> loadedPage =  
        SlottedPage::deserialize(file, page_itr);  
    pages.push_back(std::move(loadedPage));  
}
```

Flushing Database File



buzzDB

- BuzzDB saves changed pages to disk
- Flush method plays a critical role

```
void SlottedPage::flush(std::fstream &file, uint16_t page_id) const {  
    size_t page_offset = page_id * PAGE_SIZE;  
    file.seekp(page_offset, std::ios::beg);  
    file.write(page_data.get(), PAGE_SIZE);  
    file.flush(); // Ensure data is written to disk  
}
```


Inserting Tuples



buzzDB

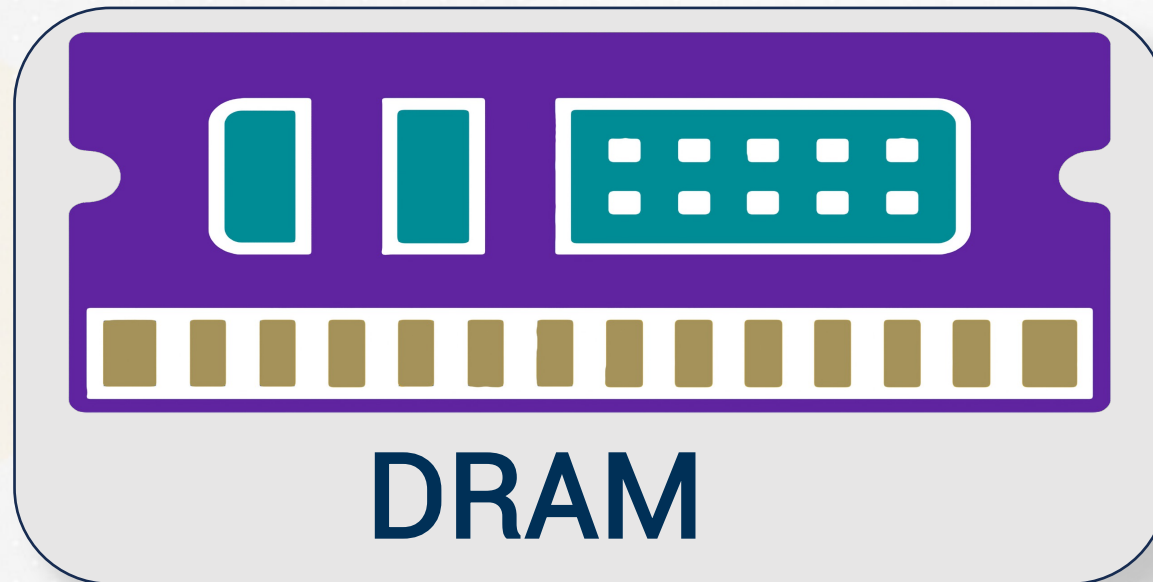
- Flush method ensures that updated SlottedPage is written to disk

```
bool try_to_insert(int key, int value) {  
    ...  
    status = pages[page_itr]->addTuple(std::move(newTuple));  
    if (status == true) {  
        pages[page_itr]->flush(file, page_itr);  
        break; // Successfully inserted and persisted the tuple  
    }  
}
```

Flushing Database File



C++ employs buffered I/O to enhance file operations' efficiency

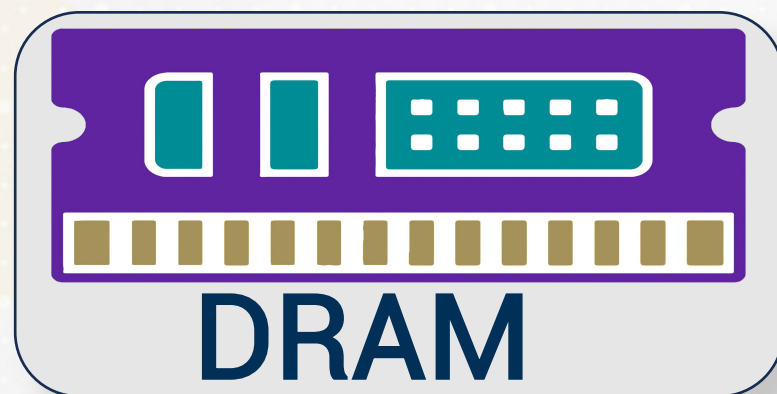


`std::ofstream` temporarily places data into an in-memory output buffer to optimize disk I/O operations

Flushing Database File



Explicitly calling `flush()` immediately writes all buffered output data to the file, an essential step for preventing data loss.



Conclusion

- Tuple deletion
- Slotted Page
- Database File Management
- File Management
- Index Construction