Lecture 5:
Slotted Page

Logistics

- Point Solutions App
  - Session ID: **database**
- Programming assignment 1 due on **Sep 7** (Gradescope)
- One-page intro sheet due on **Sep 7** (Gradescope)

Recap

- Smart Pointers
- Smart Field
- Heap vs Stack
- Page class

Lecture Overview

- Simplifying serialization
- Static method for deserialization
- Tuple deletion
- Slotted Page
- Buffer Management

Page Serialization

Simplifying Page Serialization

Page Class Serialization

Field Class Serialization

```
void Field::serialize(std::ofstream& 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()) << '
';
    }
}
```

Simplifying Page Serialization

Page Class Serialization

Tuple Class Serialization (v9)

```
void Tuple::serialize(std::ofstream& out) {
    out << fields.size() << ' ';
    for (auto& field : fields) {
        field->serialize(out);
    }
}
```

Simplifying Page Serialization

Page Class Write Function Simplification

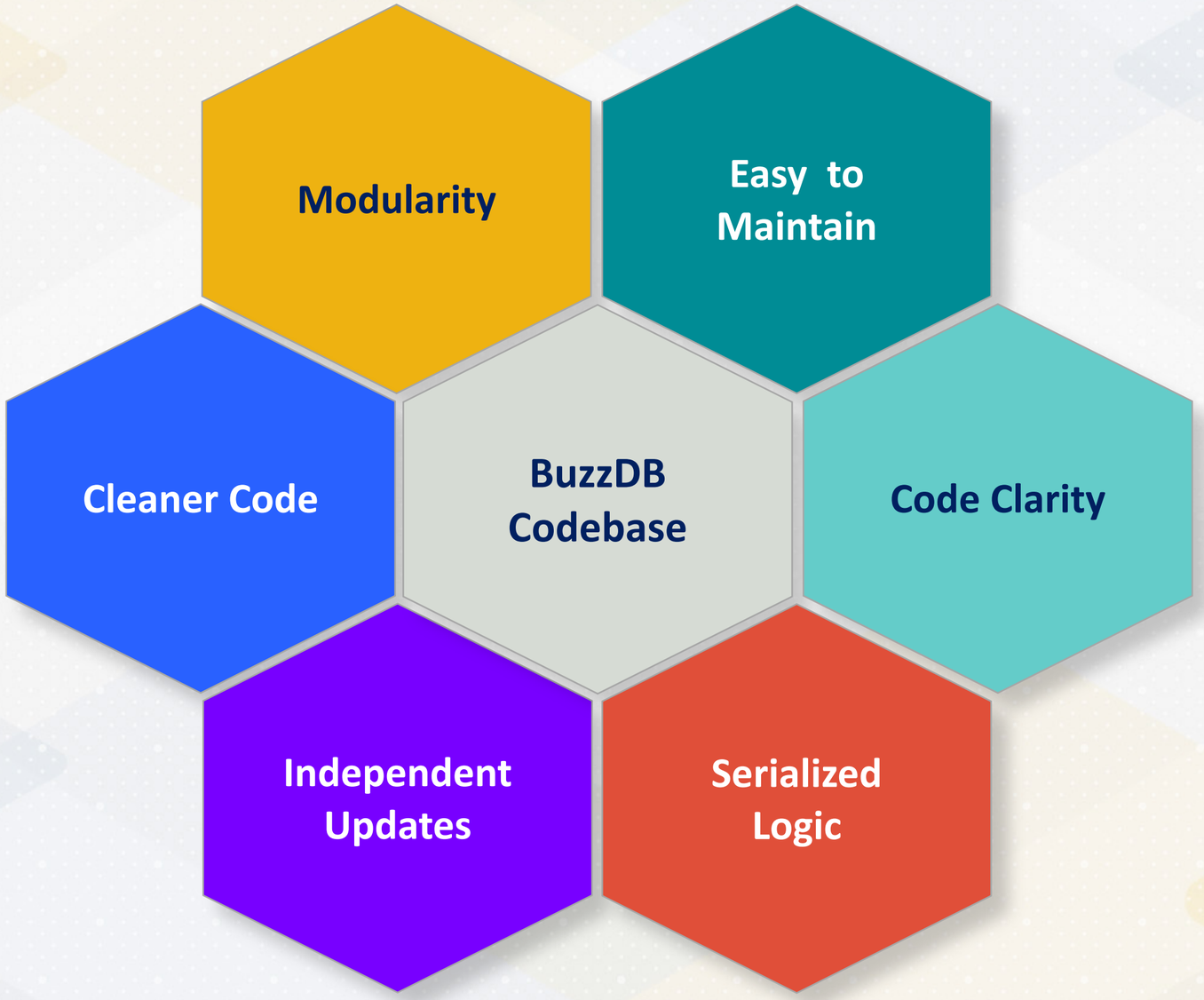
Manage Tuple Serialization

Serialize Page Tuples

Loop through Tuples

```
void Page::write(const std::string& filename) const {
    std::ofstream out(filename);
    // Serialize the number of tuples in the page
    out << tuples.size() << '\n';
    // Loop through each tuple and serialize its data
    for (auto& tuple : tuples) {
        tuple->serialize(out);
        out << '\n'; // Delimiter for each tuple
    }
    out.close();
}
```

Modular Approach to Serialization



Static Method for Deserialization

Instance Method for Deserialization

Deserialization Steps

Create New Page Object

Populate with File Data

```
// Serialize to disk
db.page.write(filename);

// Deserialize from disk
Page page2;
page2.read(filename);
```

Static vs Instance Methods in C++

Static Methods

- Pertains to Whole Class
- Invoke without Class Object Creation

Instance Methods

- Operate/Modify Data
- Applies to Specific Class Instances

Page::deserialize(filename)

- Allows for direct loading of Tuples from disk data without needing a Page object, simplifying the process of loading an on-disk page.

Direct Tuple Loading

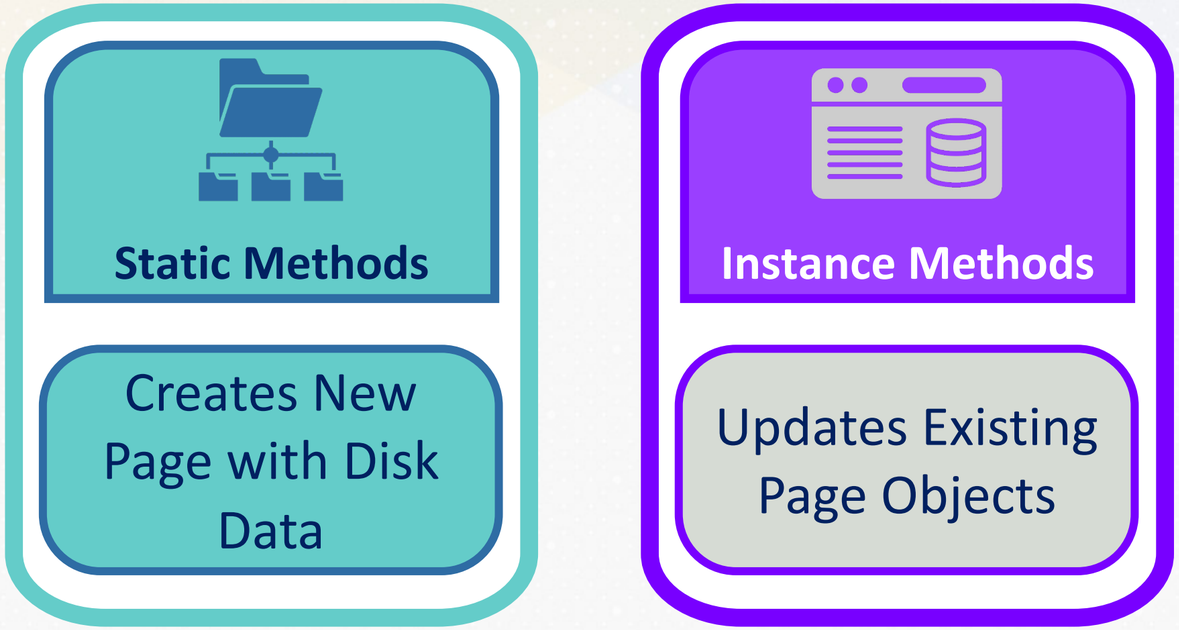
No Page Object Obligation

Streamlines Deserialization

Page State Stored Complete

```
auto loadedPage = Page::deserialize(filename);

// Deserialize from disk
// Page page2;
// page2.read(filename);
```

Static and Instance Methods




```
// Read this page from a file.
static std::unique_ptr<Page> deserialize(const
std::string& filename) {
        std::ifstream in(filename);
        auto page = std::make_unique<Page>();
        …
        return page;
}
```

Tuple Deletion

Tuple Deletion

Tuple Removal Process

- Implement deleteTuple
- Vector Bounds Checked
- Error Printed before Exit
- Page Size Use Decreased

```
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 Remove Tuples

Simulate Data Lifestyle Management

Deletion Skipped for 100 Attempts

Periodic Database Cleanup

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

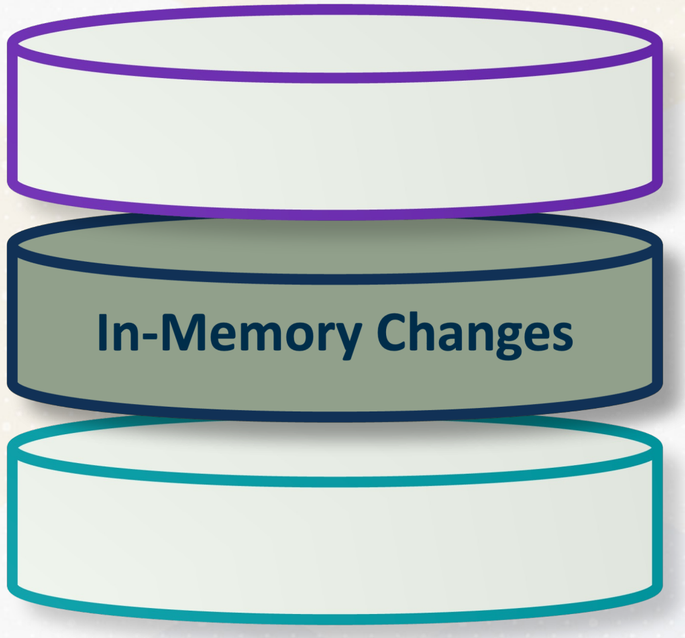
Data Inconsistency

Errors in Tuple Deletion

- Change Delay on Disk
- Re-Deserialization 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 Desynchronized
- Current and Stored State Discrepancy

Data Synchronization


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


loadedPage->write(filename);


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

Data Synchronization

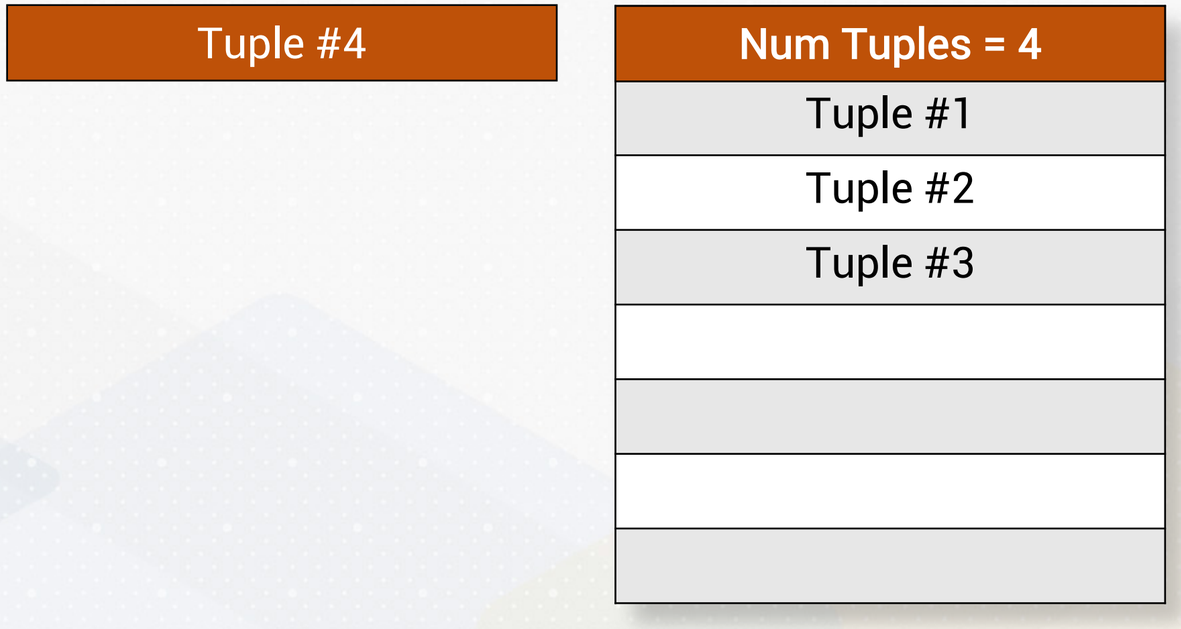
Modifications Reflected with Disk Re-Serialization

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


loadedPage->write(filename);


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

Slotted Page

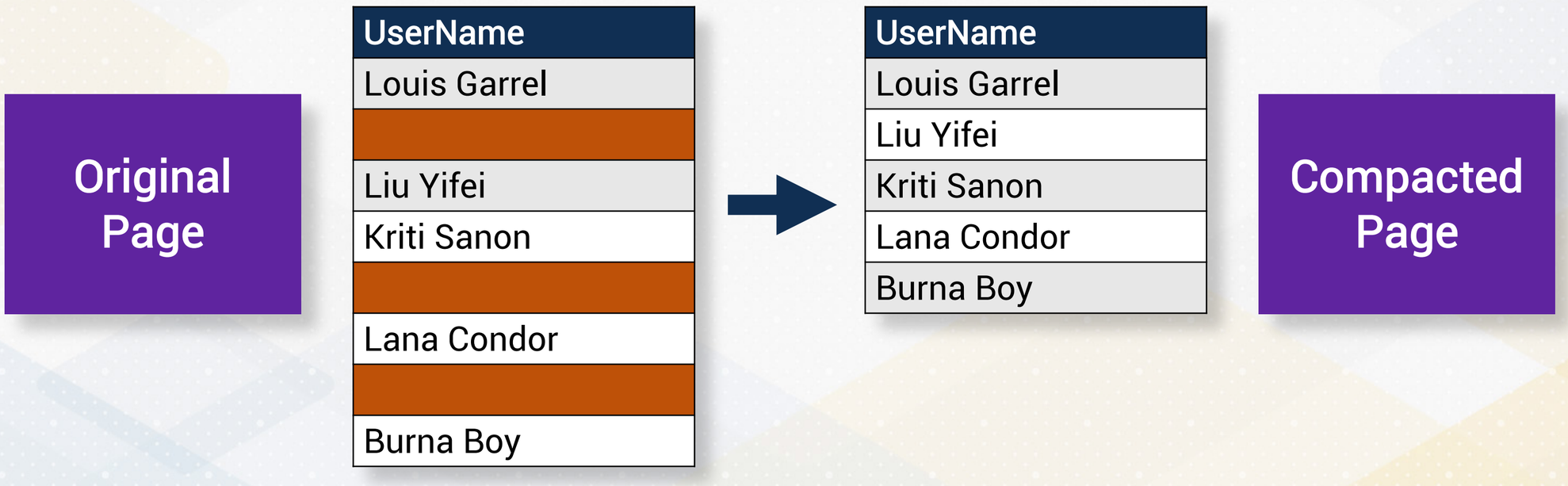
Limitations of Page Class

- **Page class:** finding space for a new record requires a linear scan through the page to identify a suitable empty tuple slot.
- Poor handling of variable-length records leads to wasted space.



Limitations of Page Class

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



Slotted Page

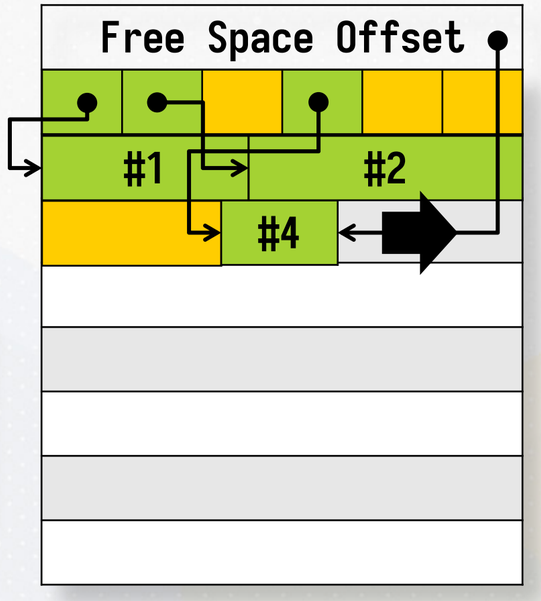
The **Slotted Page class** is a flexible structure for managing tuples

Slots are metadata entries in a page's "header" section that keep track of tuples

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

Slotted Page

Slots are metadata entries in a page's "header" section that keep track of tuples 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
    }
}
```

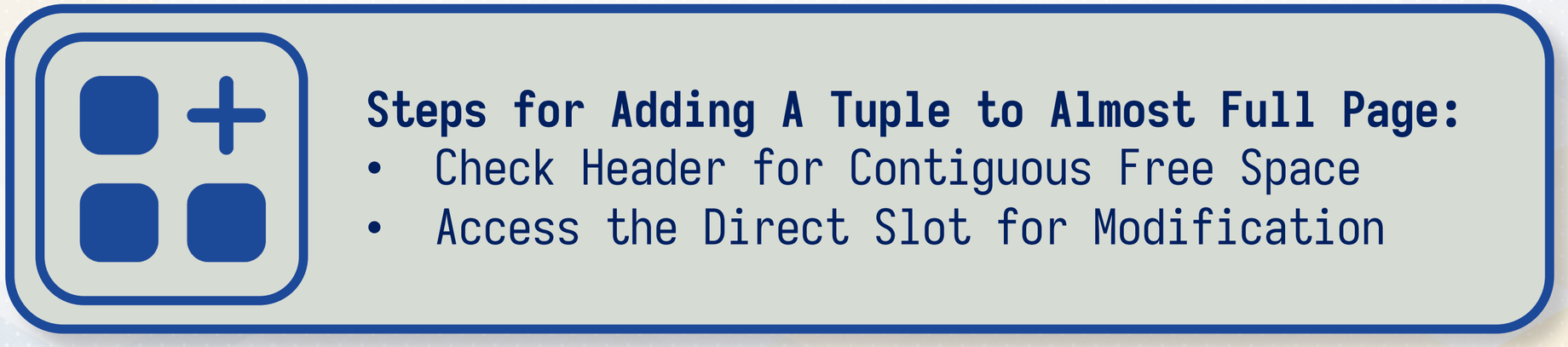
- Marks a tuple as deleted within the **Slotted Page** by updating the slot array
- Frees space for future insertions
- Efficient Space Reclamation
- Simplified Space Compaction

Slotted Page: Updating Tuple

- **Updates** that change tuple size:
  - system can either adjust 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 Header for Contiguous Free Space
- Access the Direct 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.

Buffer Management

Database File Management

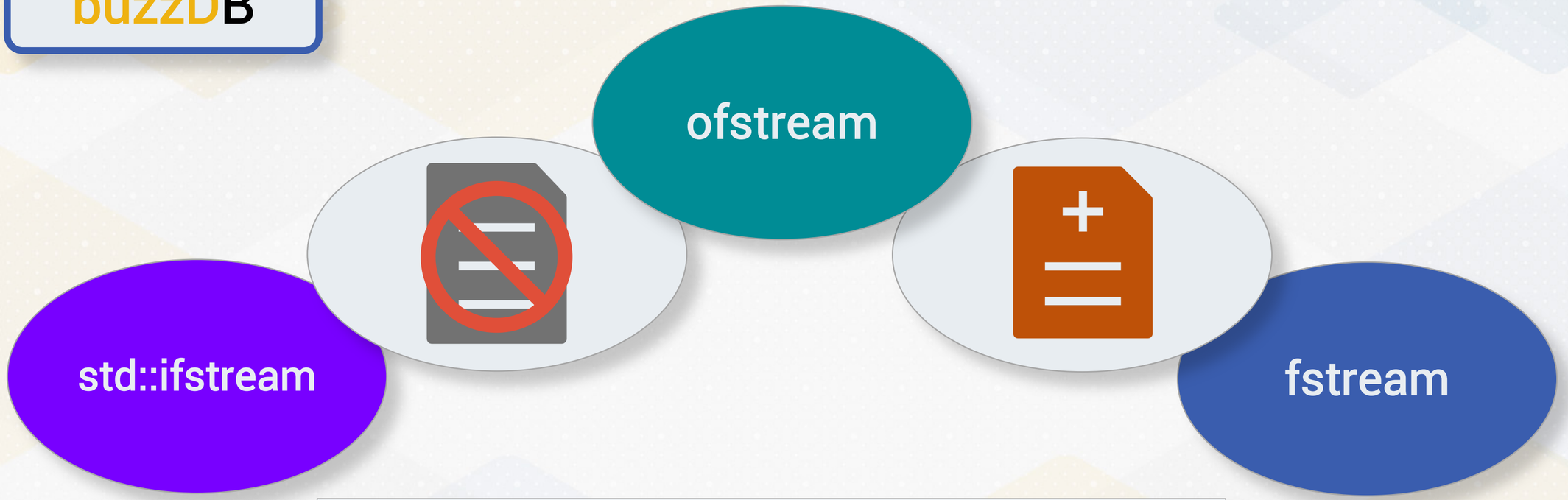
buzzDB

- 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

buzzDB




```
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

ios::in

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.

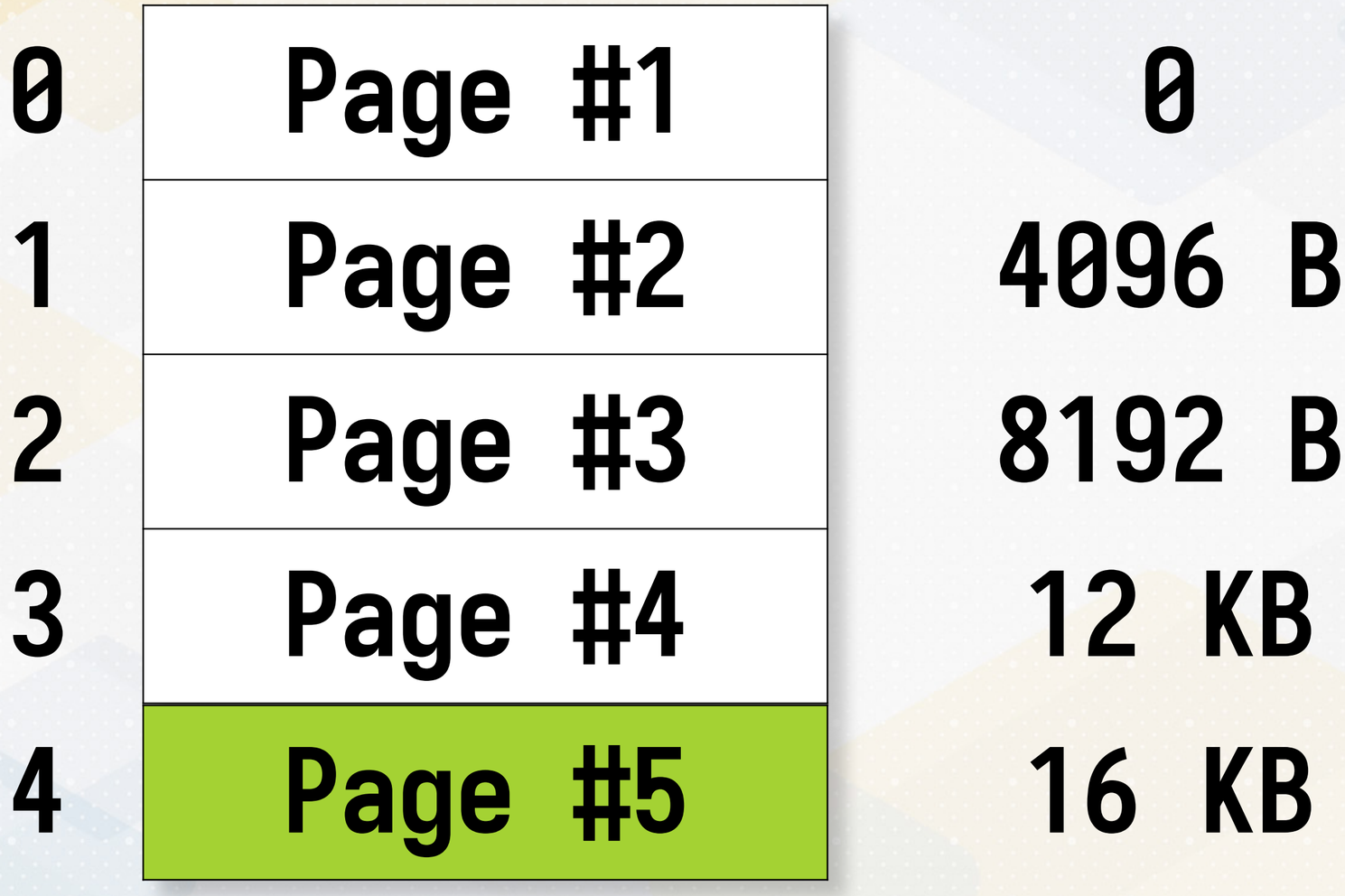
```
if (num_pages == 0) {
    extendDatabaseFile();
}

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



Inserting Data into Database File

buzzDB

- BuzzDB assesses slot usage 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 object stored in 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 changes to disk with each update
- Flush method plays 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 immediately
}
```

Inserting Tuples

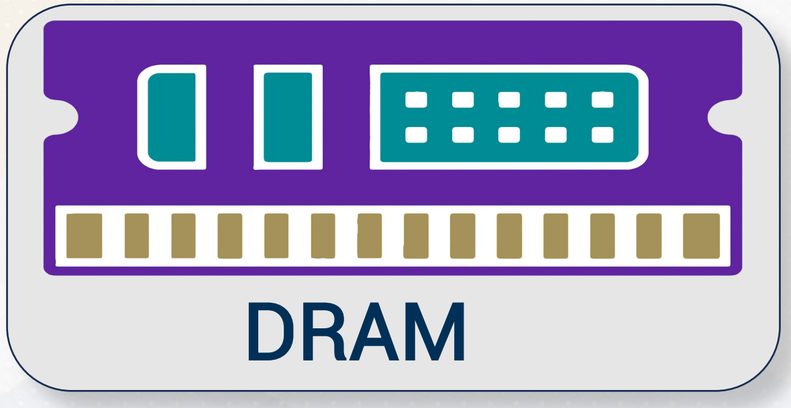
buzzDB

- Inserts tuple into the current **SlottedPage**
- Flush method records **SlottedPage** current state

```
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
        }
}
```

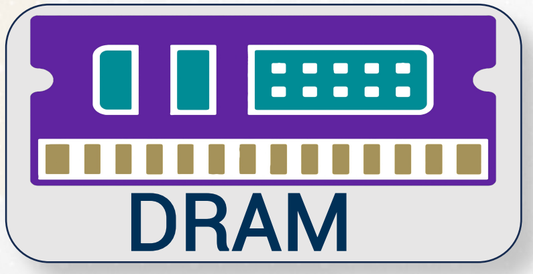
Flushing Database File

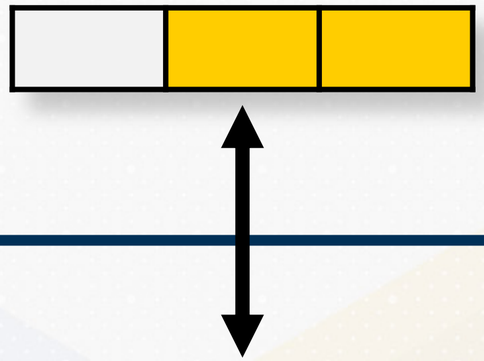
- C++ employs buffered I/O to enhance file operations' efficiency.
- Writing to a file via **std::ofstream** temporarily places data into an in-memory output buffer to optimize disk I/O operations.

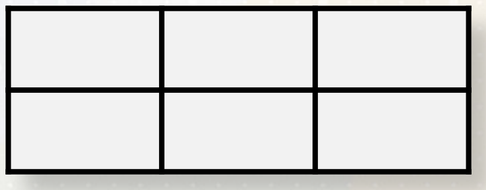


Flushing Database File

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







Conclusion

- Simplifying serialization
- Static method for deserialization
- Tuple deletion
- Slotted Page
- Buffer Management