Lecture 6: File and Storage Management

Logistics

- Point Solutions App
  - Session ID: **database**
- Programming assignment 1 due on **Sep 7** (Gradescope)
- One-page intro sheet due on **Sep 7** (Gradescope)
- Programming assignment 2 and exercise sheet 1 will be released soon.

Recap

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

Lecture Overview

- File Management
- Index Construction
- Casting in C++
- Streams in C++
- Storage Manager

File Management


Database File Management

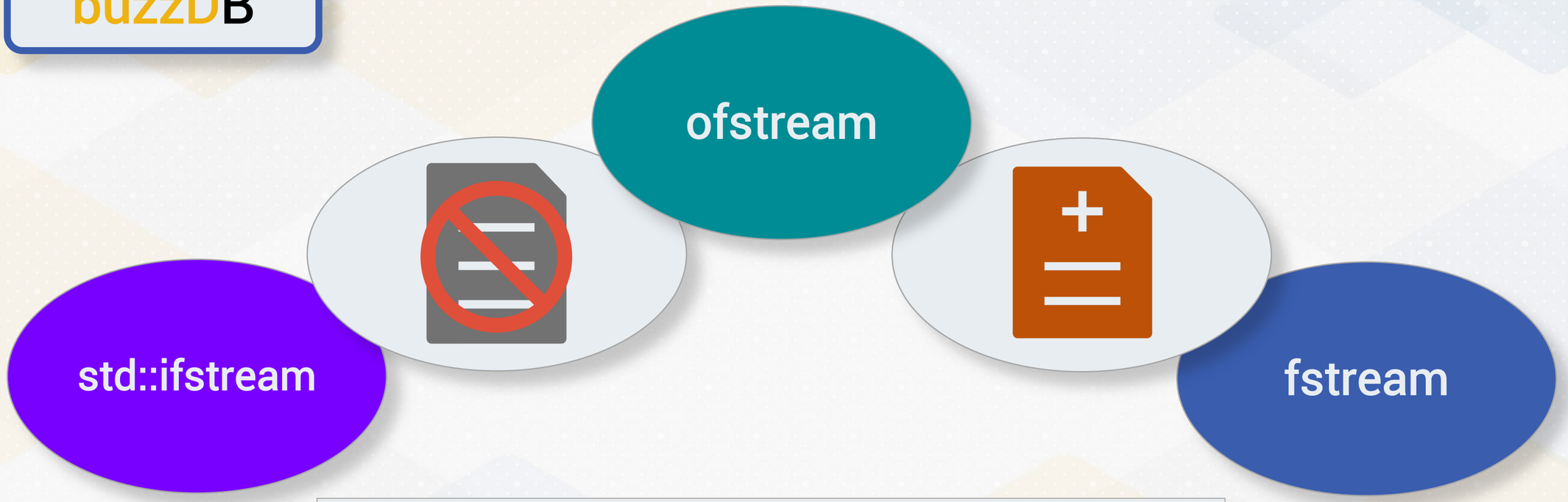
buzzDB

- 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

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.

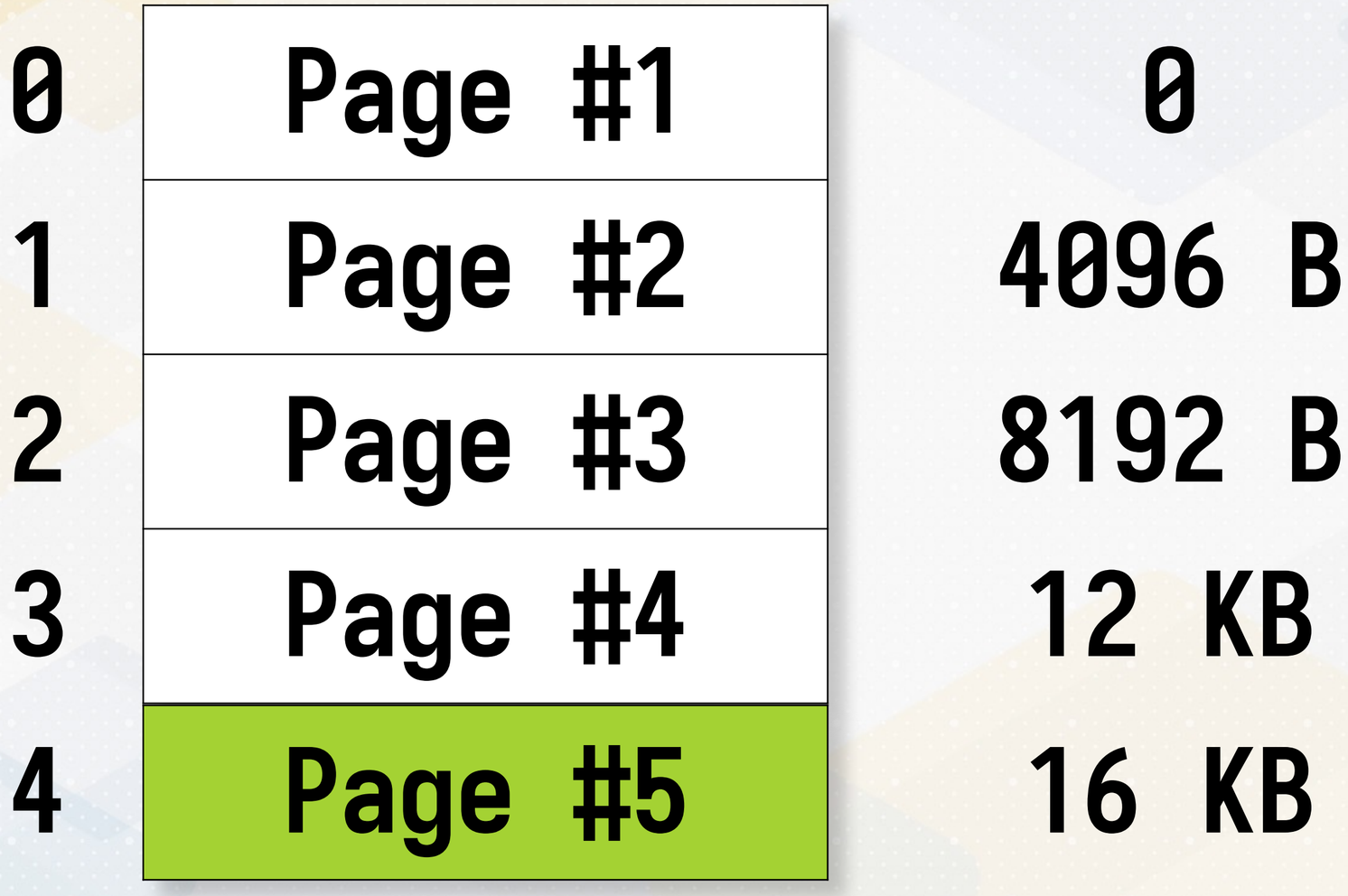
```
if (num_pages == 0) {
    extendDatabaseFile();
}

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

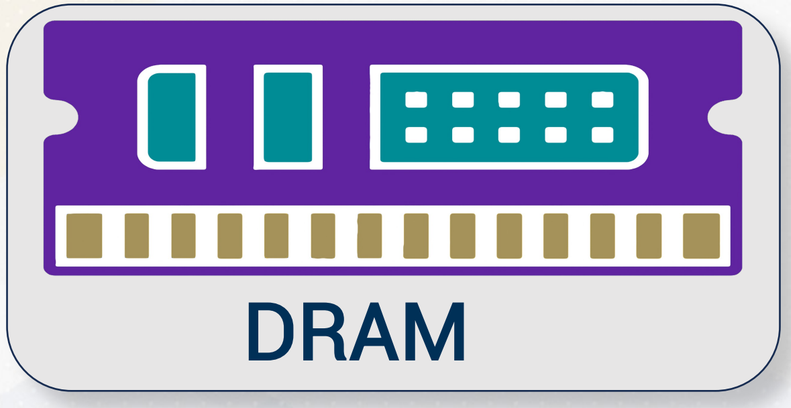
buzzDB

- 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
        }
}
```

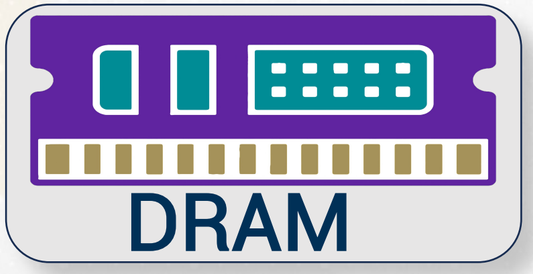
Flushing Database File

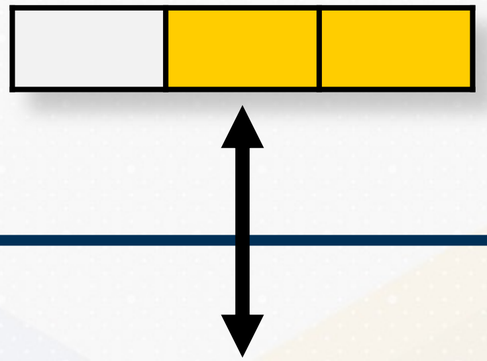
- C++ employs buffered I/O to enhance file operations' efficiency.
- Writing to a file via **std::ofstream** temporarily places data into an in-memory output buffer to optimize disk I/O operations.

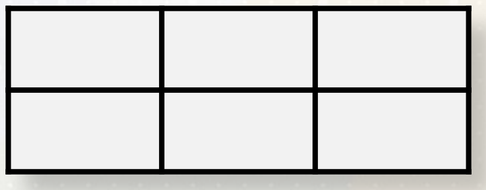


Flushing Database File

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







Index Construction

Index

UNORGANIZED LIBRARY

Book ID

	Author
0	Orwell
1	Austen
2	Austen
3	Hobbes
4	Orwell
5	Orwell
6	Hobbes
7	Austen

LIBRARY INDEX

Austen	1, 2, 7
Hobbes	3, 6
Orwell	0, 4, 5

Index Construction

- Index based on the **on-disk database file**
- Index removes table scan requirement for answering the **aggregate query**

```
void BuzzDB::scanTableToBuildIndex() {
    for (size_t page_itr = 0; page_itr < num_pages; page_itr++)
{
        char* page_buffer = pages[page_itr]->page_data.get();
        Slot* slot_array = reinterpret_cast<Slot*>(page_buffer);
        for (size_t slot_itr = 0; slot_itr < MAX_SLOTS;
slot_itr++) {
            // Build index using the tuple stored in the slot
            }
        }
    }
}
```

Index Construction


```
void BuzzDB::scanTableToBuildIndex() {
    for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {
        char* page_buffer = pages[page_itr]->page_data.get();
        // Build index using tuples stored in the page
        }
    }
}
```

Index Construction

Step 2: Cast **page buffer** to Slot array to access slot metadata.
Step 3: Loop over each slot, checking for **non-empty slots** indicating stored tuples

```
Slot* slot_array = reinterpret_cast<Slot*>(page_buffer);
for (size_t slot_itr = 0; slot_itr < MAX_SLOTS; slot_itr++) {
      if (slot_array[slot_itr].empty == false) {
```

reinterpret_cast

Type of casting that converts any pointer type into any other pointer type

Converts a **char* pointer** (page_buffer) to a **Slot* pointer**

Essential to access slots array directly from a raw memory buffer

```
Slot* slot_array = reinterpret_cast<Slot*>(page_buffer);
 for (size_t slot_itr = 0; slot_itr < MAX_SLOTS; slot_itr++) {
        if (slot_array[slot_itr].empty == false) {
          ...
        }
 }
```

Index Construction

- Extract **tuple data** from the slot's offset within the **page buffer**
- Deserialize the tuple data to retrieve **key-value pairs**
- Faster access obtained with in-memory index with key-value pairs

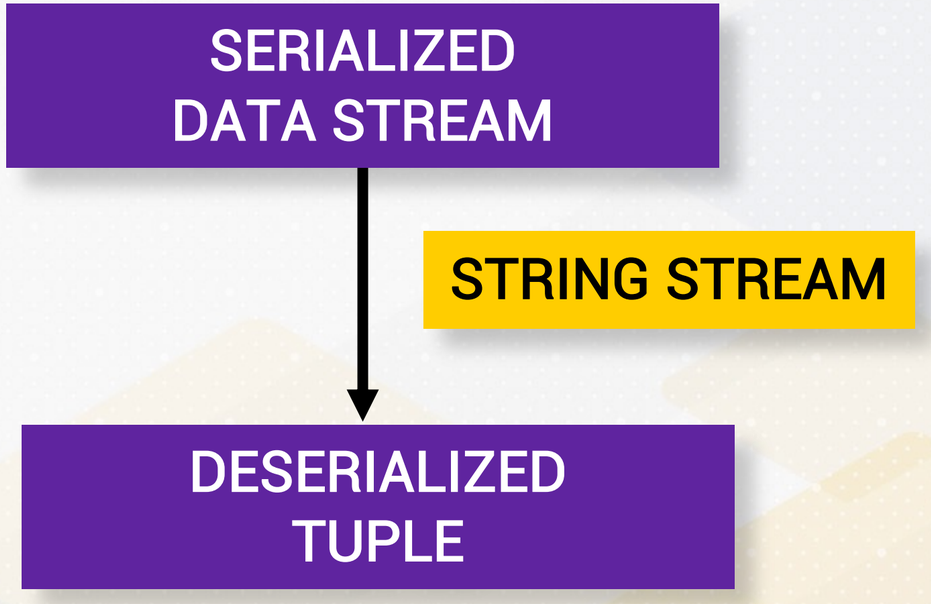
```
const char* tuple_data = page_buffer + slot_array[slot_itr].offset;
std::istringstream iss(tuple_data);
auto loadedTuple = Tuple::deserialize(iss);
int key = loadedTuple->fields[0]->asInt();
int value = loadedTuple->fields[1]->asInt();
index[key].push_back(value);
```

istringstream

istringstream is a library used to perform operations on string streams

“105 Kriti Sanon 5000”

ID	105
First Name	Kriti
Last Name	Sanon
Salary	5000



istringstream

- Initialize string with tuple data, allowing for easy **deserialization**
- Facilitates the conversion of serialized string data back into **Tuple objects**

```
const char* tuple_data = page_buffer + slot_array[slot_itr].offset;
std::istringstream iss(tuple_data);
auto loadedTuple = Tuple::deserialize(iss);
int key = loadedTuple->fields[0]->asInt();
int value = loadedTuple->fields[1]->asInt();
index[key].push_back(value);
```

Casting in C++

Casting in C++

C++ provides four main casting operators to convert data from one type to another:

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast (later)

Casting in C++: static_cast

Use it when you're converting types that are naturally compatible with each other, like integers and floats.

Example: Converting a float to an **int** to store it as a **key** in the BuzzDB index.

```
float floatKey = 123.45;
int intKey = static_cast<int>(floatKey);
// Conversion for using as a key
```

Casting in C++: reinterpret_cast

This cast transforms any pointer into any other pointer type, even if the types are unrelated.

```
char* pageData = getPageData(); // Assume this function gets raw data of a page
Slot* slots = reinterpret_cast<Slot*>(pageData); // Treat raw data as array of Slots
```

Casting in C++: const_cast

- **const_cast** modifies the **constness** of pointers and references
- Either adds or removes the **const** qualifier

```
const int* constPtr = new int(42);
int* modifiablePtr = const_cast<int*>(constPtr);
*modifiablePtr = 21; // Modifying the originally constant integer
std::cout << "Modified value: " << *modifiablePtr << std::endl;
```

Streams in C++

Stream Abstraction

Conceptual model for Handling Generalized I/O Operations

Abstracts Data Source or Destination Specifics

Stream Abstraction Focuses on **Flow of Data**

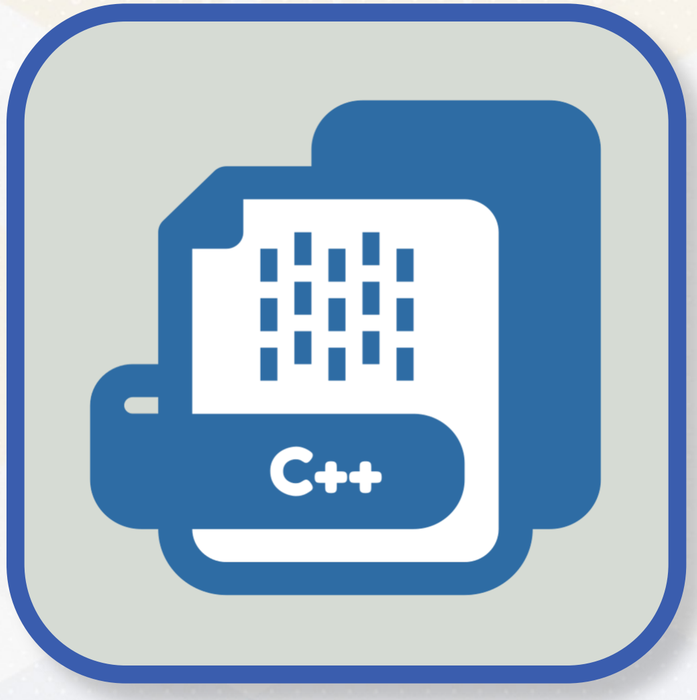
File I/O

Memory Buffer

Console I/O

Network I/O

Streams in C++



fstream for file I/O

stringstream for in-memory manipulation

File I/O with fstream

fstream | ad from/to Files

```
std::fstream file("data.txt", std::ios::in |
std::ios::out);
std::string line;
while (getline(file, line)) {
    std::cout << line << std::endl;
}
file << "New line in file\n";
file.close();
```

Used for Persistent Data Storage & Retrieval

Manipulating Strings with stringstream

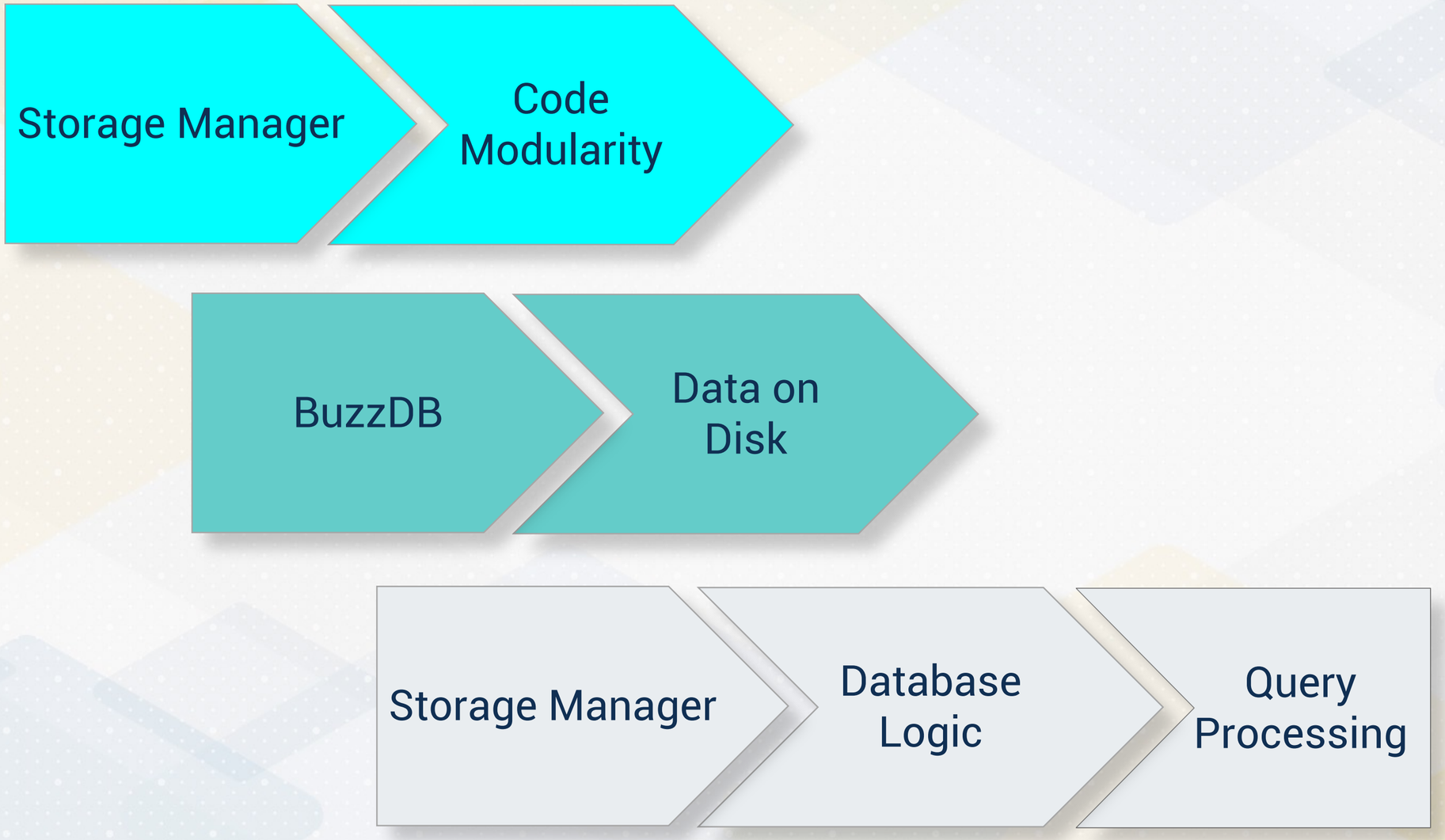
stringstream | ngs Teated
s Streams

```
std::stringstream ss;
ss << 100 << ' ' << 200; // Inserting integers into the string stream
int a, b;
ss >> a >> b; // Extracting integers back from the string stream
std::cout << "a: " << a << ", b: " << b << std::endl;
```

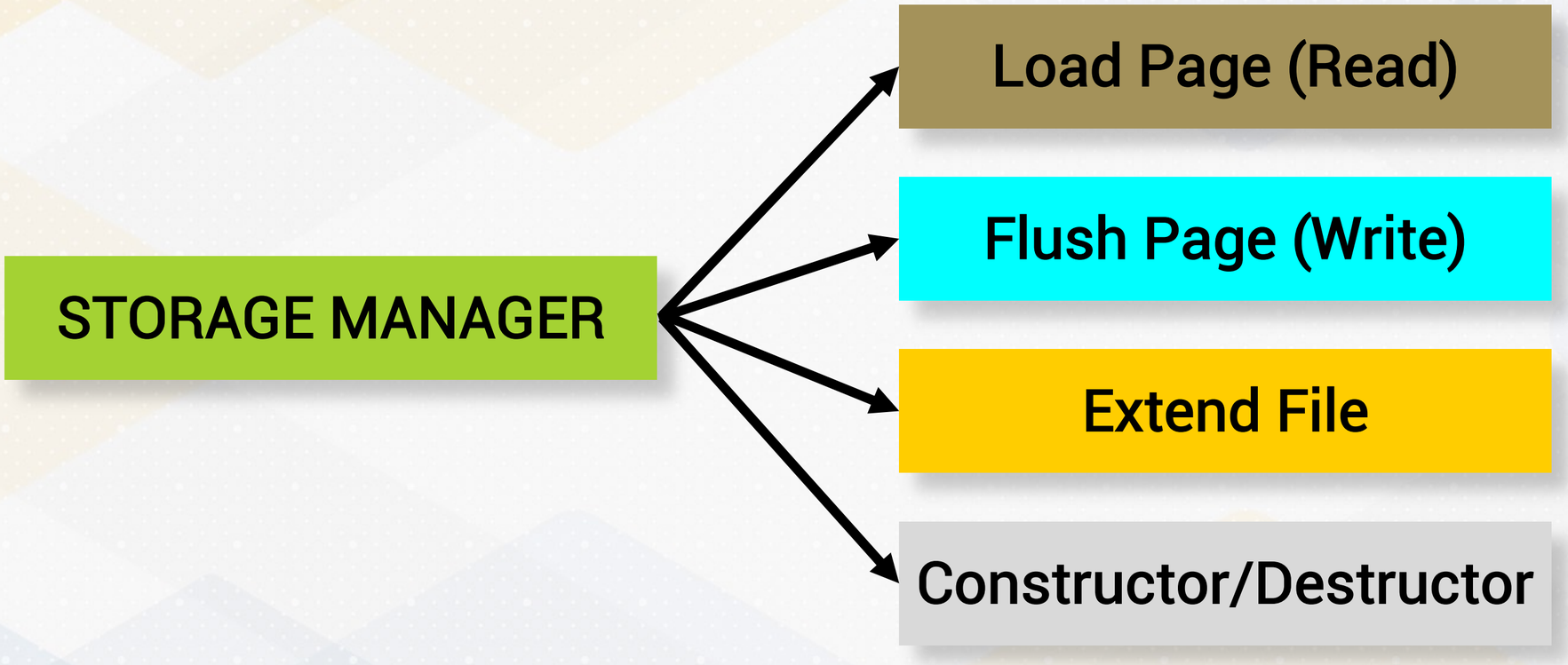
Used for Persistent Data Storage & Retrieval

Storage Manager

Storage Manager



Storage Manager



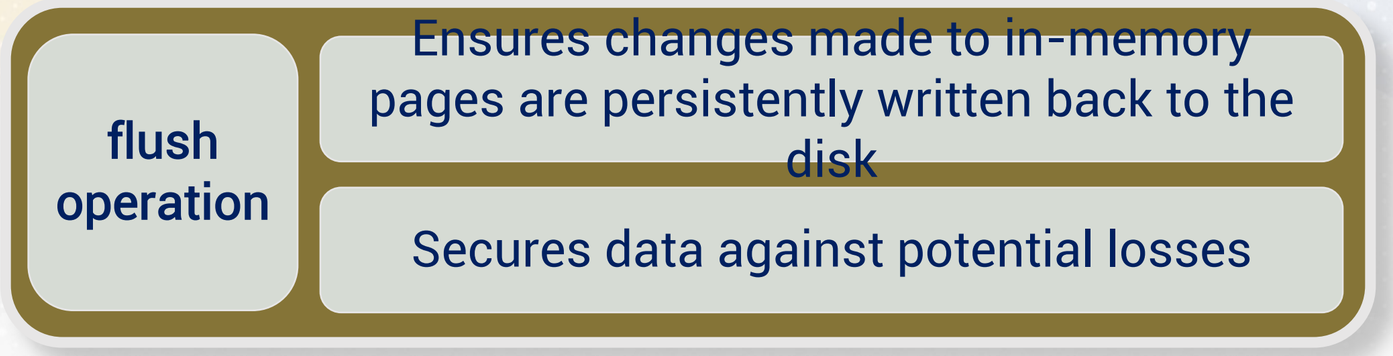
Storage Manager

```
StorageManager::StorageManager() {
    fileStream.open(database_filename,
std::ios::in | std::ios::out);
    if (!fileStream) {
        fileStream.clear(); // Reset the state
        fileStream.open(database_filename,
std::ios::out);
        fileStream.close();
    }
    fileStream.open(database_filename,
std::ios::in | std::ios::out);
    // Calculate number of pages
    fileStream.seekg(0, std::ios::end);
    num_pages = fileStream.tellg() / PAGE_SIZE;
}
```

Dynamic File Extension


```
void StorageManager::extend() {
    auto empty_slotted_page = std::make_unique<SlottedPage>();
    fileStream.seekp(0, std::ios::end);
    fileStream.write(empty_slotted_page->page_data.get(), PAGE_SIZE);
    fileStream.flush();
    num_pages += 1;
}
```

Data Persistence




```
void StorageManager::flush(uint16_t page_id) {
    size_t page_offset = page_id * PAGE_SIZE;
    fileStream.seekp(page_offset, std::ios::beg);
    fileStream.write(pages[page_id]->page_data.get(), PAGE_SIZE);
    fileStream.flush();
}
```

Data Loading


```
std::unique_ptr<SlottedPage> load(uint16_t page_id) {
    fileStream.seekg(page_id * PAGE_SIZE, std::ios::beg);
    auto page = std::make_unique<SlottedPage>();
    // Read the content of the file into the page
    if(fileStream.read(page->page_data.get(), PAGE_SIZE)){
        //std::cout << "Page read successfully from file." << std::endl;
    }
    else{
        std::cerr << "Error: Unable to read data from the file. \n";
        exit(-1);
    }
    return page;
}
```

Storage Manager Destructor

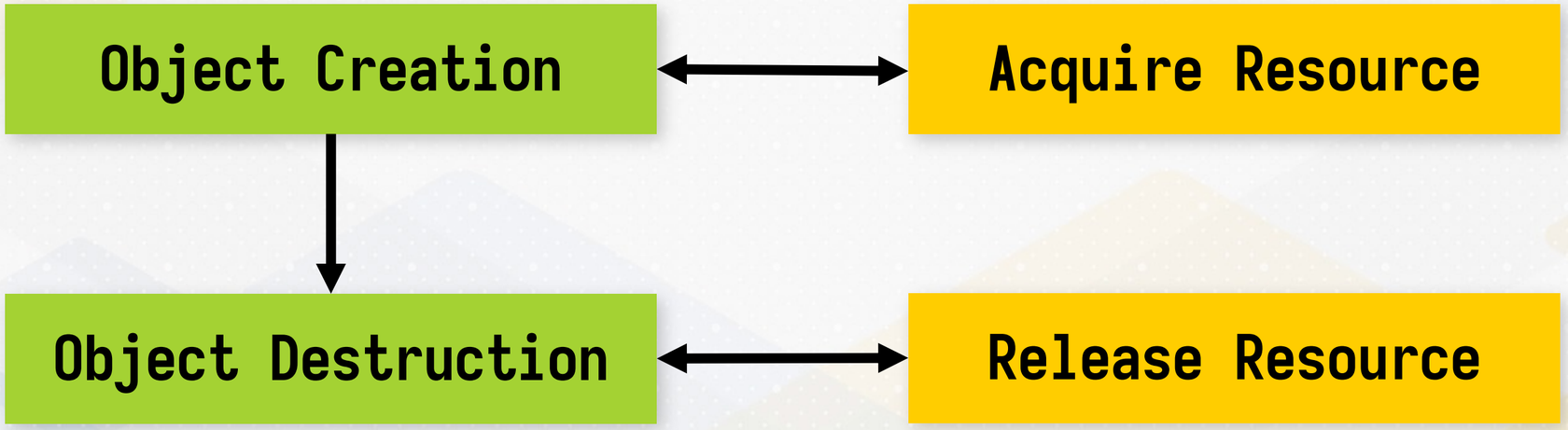
Destructor ensures that the open file stream is correctly closed and cleaned up

```
StorageManager::~StorageManager() {
    if (fileStream.is_open()) {
        fileStream.close();
    }
}
```

RALL in C++

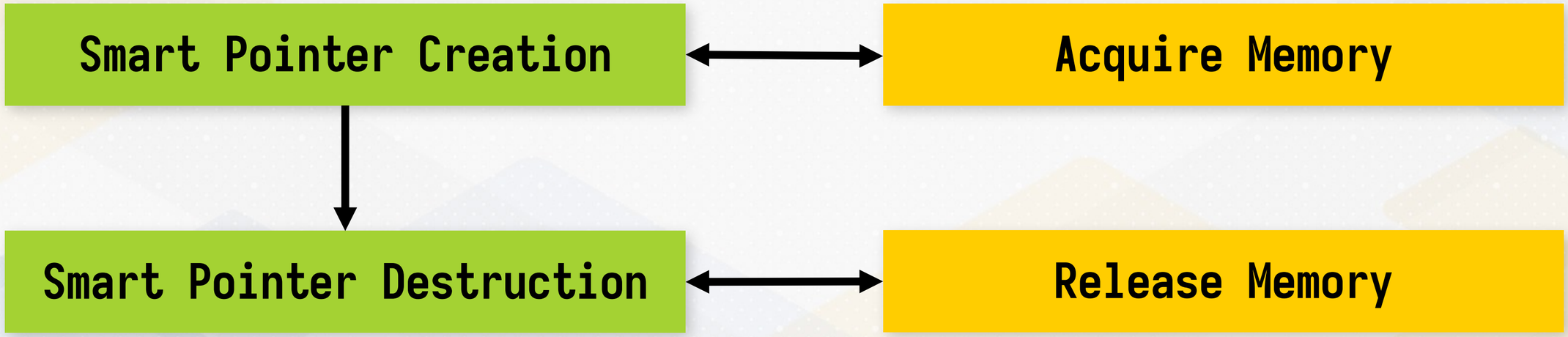
RAII (Resource Acquisition Is Initialization)

a programming idiom that binds the lifecycle of resources (i.e., file handles and memory) to object lifetimes



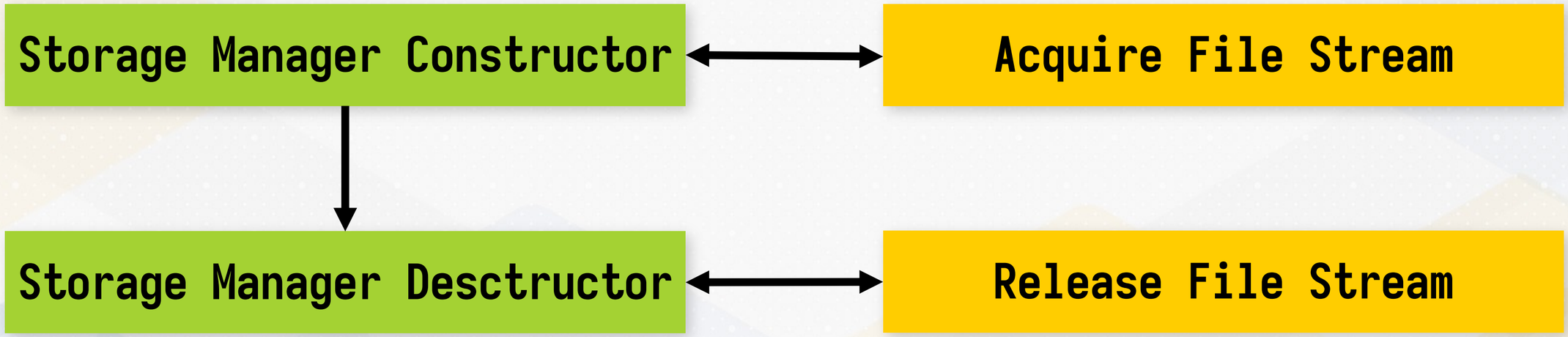
RALL in C++: Smart Pointers

- **Smart pointers** illustrate the RAII principle
- Resource lifetime (memory) is tied to the scope of the smart pointer, ensuring **deterministic** and **automatic** resource cleanup



RALL in C++: File Handles

The **Storage Manager** also illustrates the RAII principle.



Integration into BuzzDB

buzzDB

Utilizes the **Storage Manager** to streamline the process of inserting tuples into the database

```
bool BuzzDB::try_to_insert(int key, int value) {
    for (size_t page_id = 0; page_id < sm.num_pages; ++page_id) {
        auto& page = sm.pages[page_id]; // Access page via
Storage Manager
        if (page->addTuple(/* tuple data */)) {
            sm.flush(page_id); // Persist changes to disk
            return true; // Successful insertion
        }
    }
    return false; // Indicate failure to insert due to full pages
}
```

Conclusion

- File Management
- Index Construction
- Casting and Streams in C++
- Storage Manager