



Lecture 6: File and Storage Management



Logistics

- Point Solutions App
 - Session ID: **database**
- Programming assignment 1 due on **Sep 10** (Gradescope)
- One-page intro sheet due on **Sep 10** (Gradescope)
- Programming assignment 2 and exercise sheet 1 will be released soon.

Recap

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

Lecture Overview

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

File Management



Database File Management

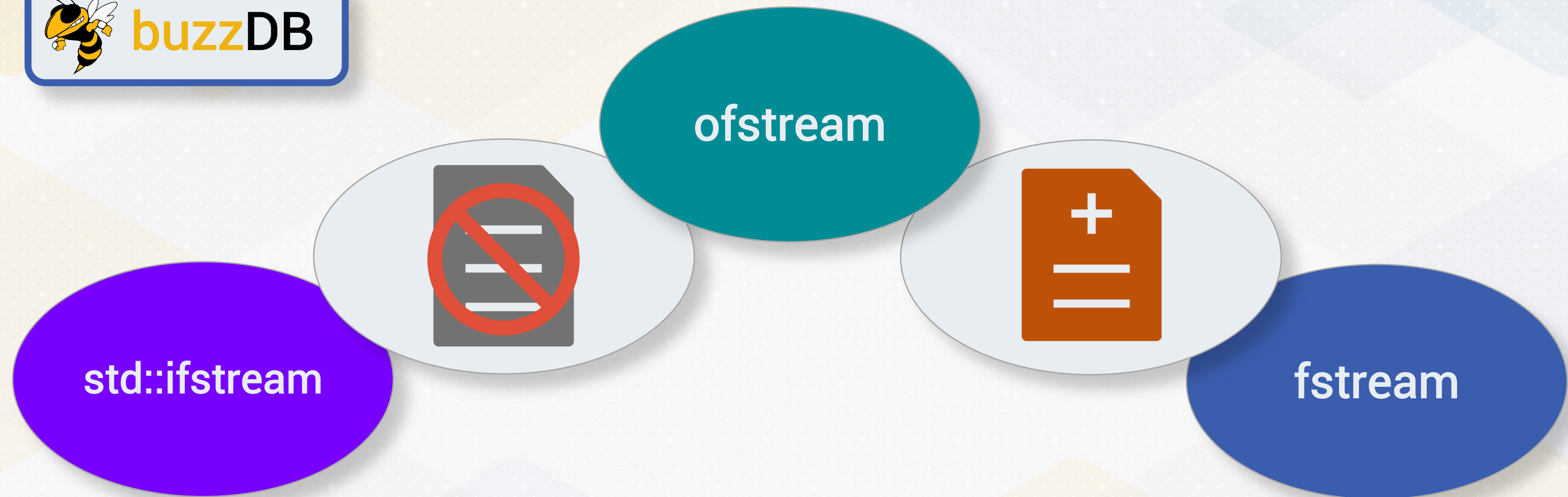


buzzDB

- 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

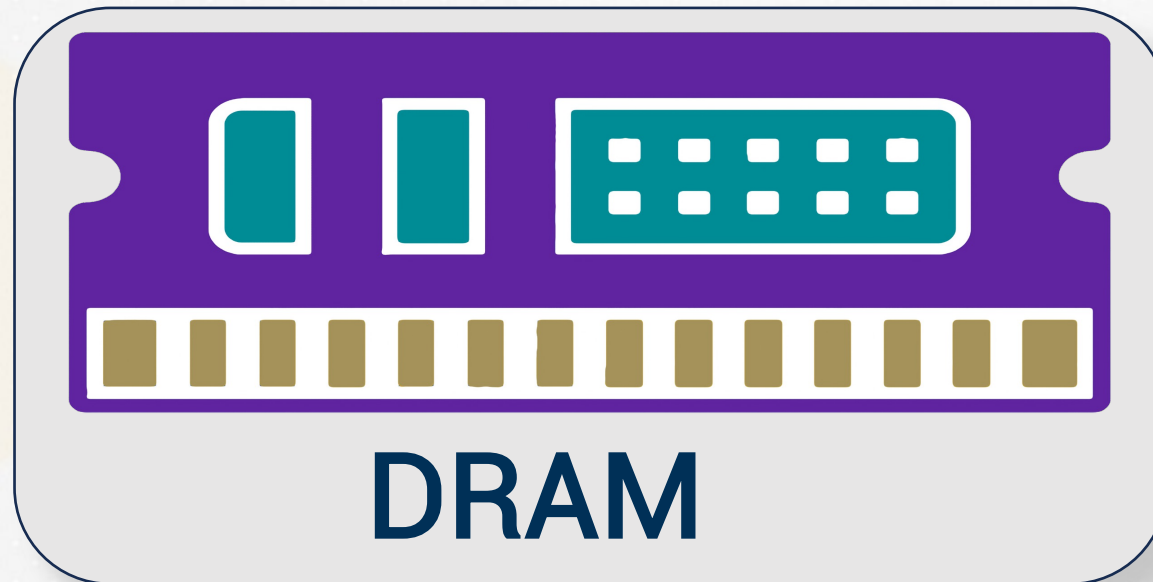
- Flush method ensures that updated SlottedPage is written to disk

```
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

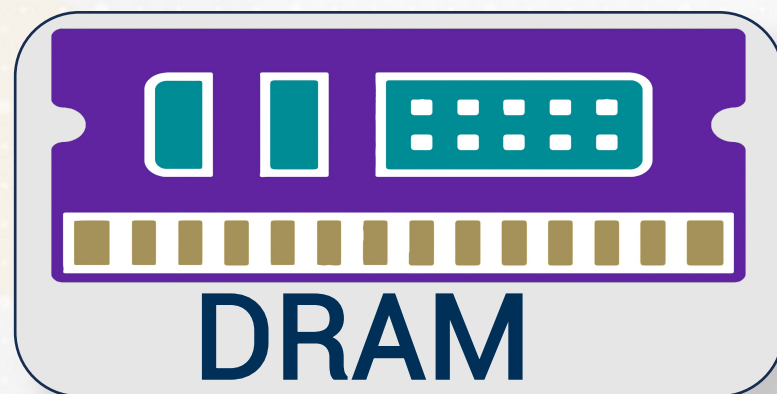


`std::ofstream` temporarily places data into an in-memory output buffer to optimize disk I/O operations

Flushing Database File



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



Index Construction



What is an Index?

UNORGANIZED LIBRARY

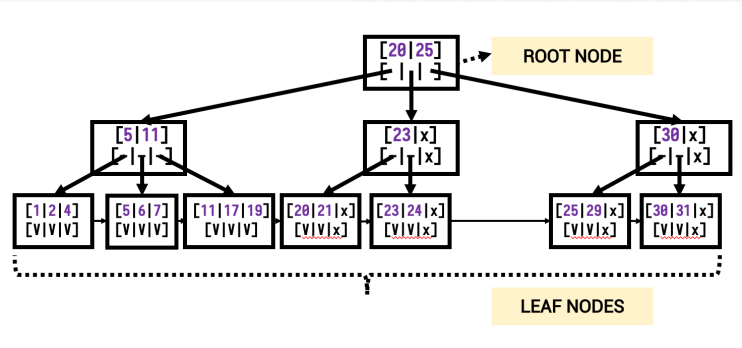
Book ID

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

Author

LIBRARY INDEX

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



TREE DATA STRUCTURE

Index Construction



- Build index using the on-disk database file
- Iterate over all the pages in the file and all the slots in each page

```
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



Step 1: Obtain a pointer to the page data

```
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);
```

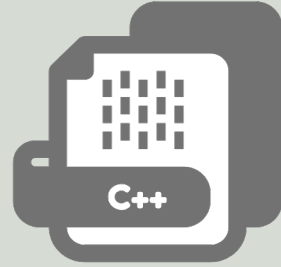
Index Construction



- Extract tuple data from the slot's offset within the page buffer
- Deserialize the tuple data to retrieve **key-value pairs**
- Add key-value pairs to the index

```
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"

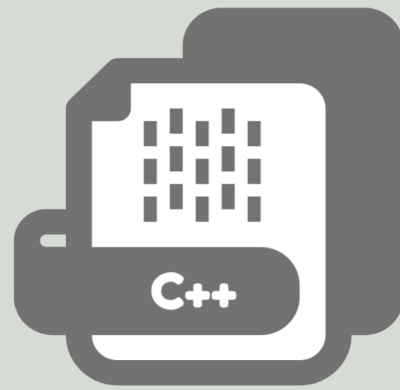
ID	105
First Name	Kriti
Last Name	Sanon
Salary	5000

**SERIALIZED
DATA STREAM**

STRING STREAM

**DESERIALIZED
TUPLE**

istringstream



- Initialize string with tuple data for easy deserialization
- Convert 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);
```




Case Study: Postgres Fsync Issue



Postgres Fsync Issue

- Postgres relies on the OS for disk writes, consistent with its “database only” philosophy.
- Two types of writes:
 - **Data file** → always go through the kernel cache.
 - **Logfile** → can use direct I/O or kernel cache

Postgres Fsync Issue

- The issue arises at the kernel level when flushing cache to disk with fsync.
- Postgres assumes the OS will retry flushing if fsync fails, but this is incorrect.
- When fsync fails, cached data is discarded and cannot be retrieved.
- Behavior differs slightly by filesystem (ext4, xfs, btrfs, etc.), but the outcome is the same: no retry of the failed flush.

Postgres Fsync Issue

- Postgres expected fsync errors to be visible across all file descriptors for a file.
- In reality, error reporting is inconsistent:
 - On Linux < 4.13: sometimes no process sees the error.
 - On newer kernels: only the first process sees it, others don't.
- Actual behavior depends on the OS kernel version.

Casting in C++



Casting in C++

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

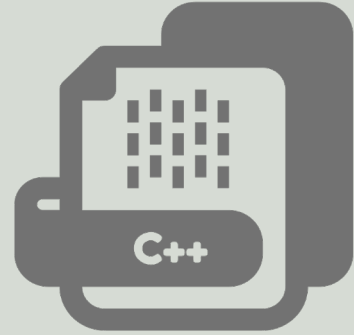
`static_cast`

`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.

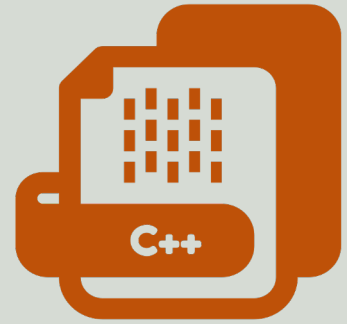
```
float floatKey = 123.45;  
int intKey = static_cast<int>(floatKey);  
// Conversion for using as a key
```

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
I/O operations

Abstracts
Data Source or
Data Destination
specifics

Stream
Abstraction
focuses on
Flow of Data

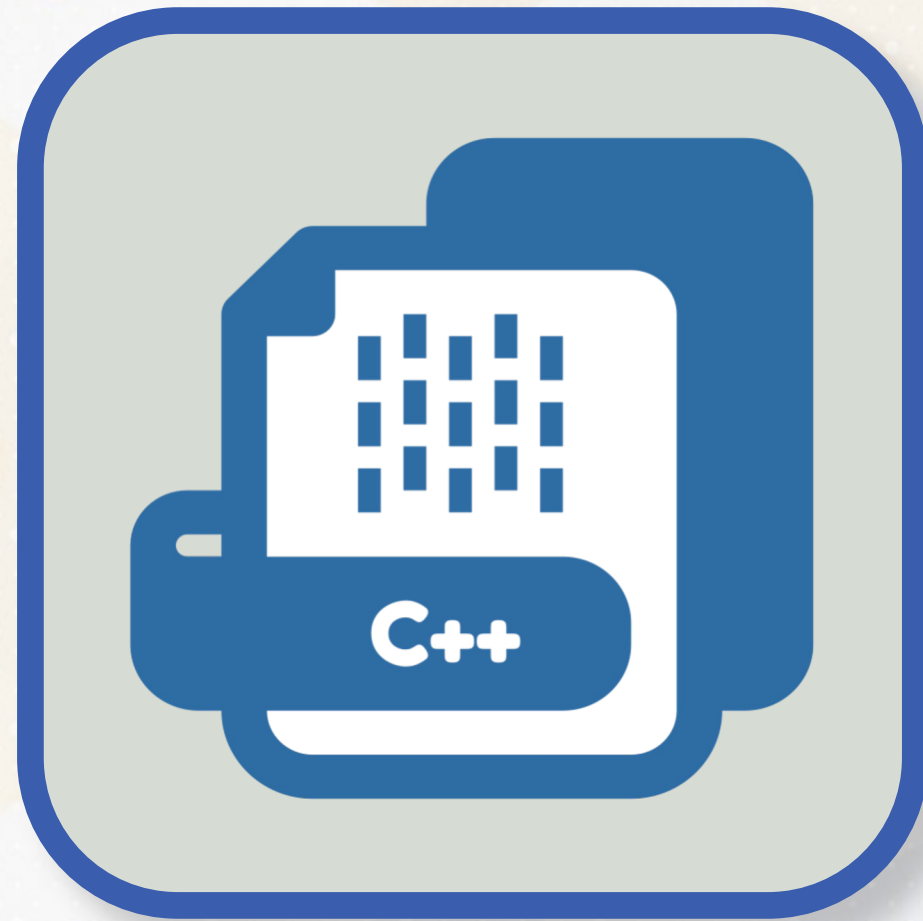
File
I/O

Memory
Buffer

Console
I/O

Network
I/O

Streams in C++



`fstream` for
file I/O

`stringstream` for
in-memory string manipulation

File I/O with fstream

fstream

Read 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

Strings treated
as 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 converting data to and from strings

Storage Management

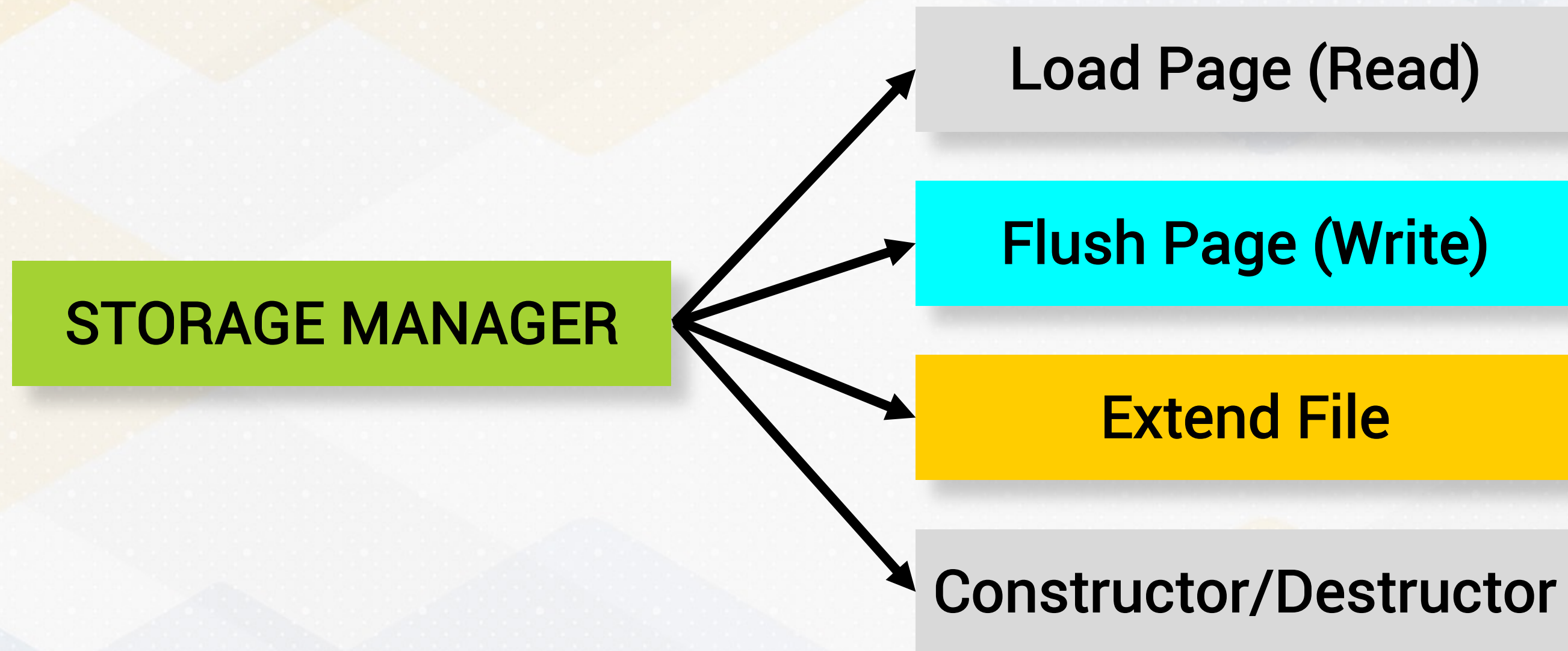


Storage Manager

STORAGE MANAGER

BUZZDB: QUERY PROCESSING

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



Database dynamically extended by
adding new pages as needed

```
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

**flush
operation**

Ensures changes made to in-memory
pages are written back to the disk

Secure data against potential data loss

```
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



Load method reads a page
from the database file

```
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 opened file stream is closed

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

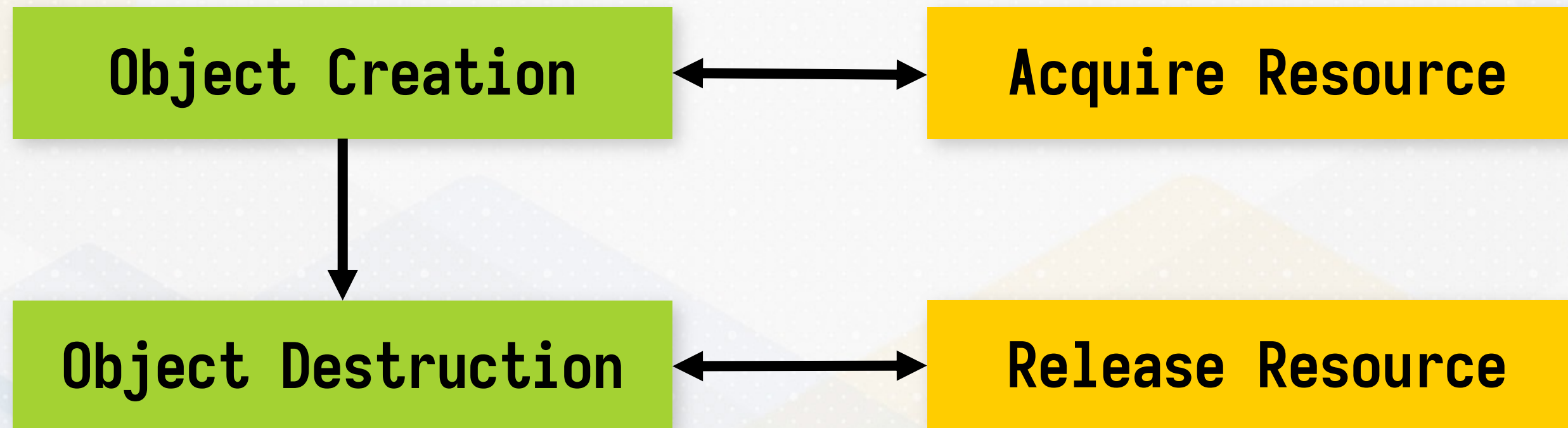

RAII in C++



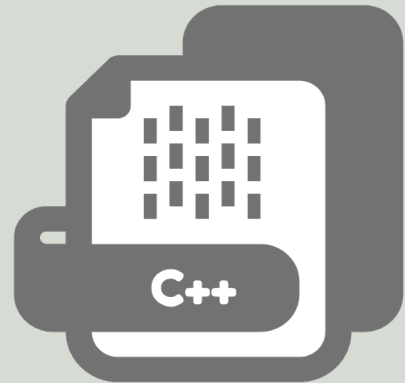
RAII in C++

RAII (Resource Acquisition Is Initialization)

A programming idiom that binds the lifecycle of resources (e.g., file handles) to object lifetimes



RAII in C++: Smart Pointers



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

Smart Pointer Creation

Acquire Memory

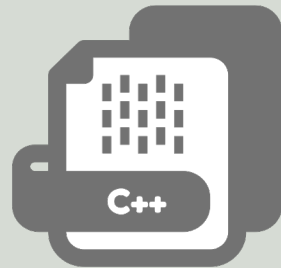


Smart Pointer Destruction

Release Memory



RAII in C++: File Handles



Storage Manager illustrates the RAII principle.

Storage Manager Constructor

Acquire File Stream



Storage Manager Destructor

Release File Stream



Integration into BuzzDB



buzzDB

Utilize the Storage Manager to read and write pages, enhancing code modularity

```
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 Management
- RAI in C++