Lecture 2:
Storage Management

Logistics

- Lecture slides posted on course website
- Office hours sign-up sheet posted on Canvas

Recap

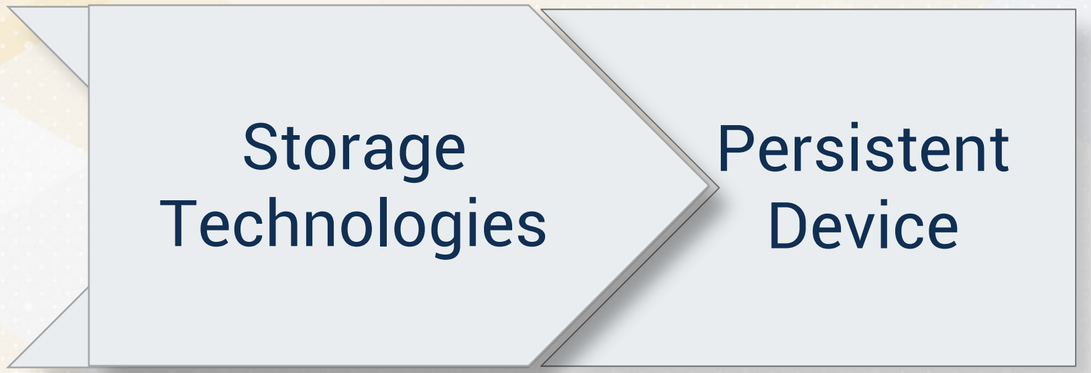
- Limitations of a Flat-File database system
- Benefits of a Relational database system
- Relational Operators and Relational Algebra
- Transactions and ACID properties

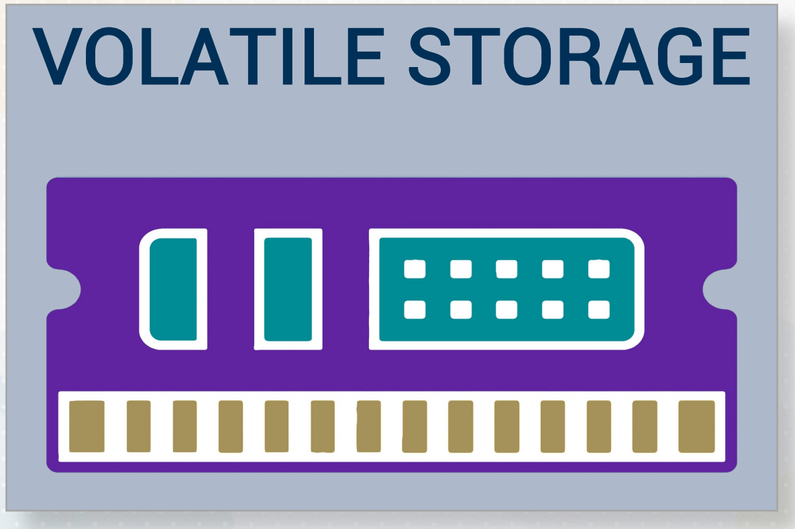
Lecture Overview

- Storage Management
- Smart Pointers and Pages
- Slotted Page
- Buffer Management
- 2Q Policy

Tuples and Fields

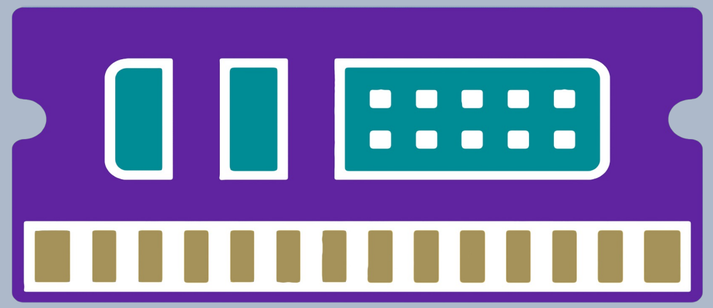
Storage Technologies





Volatile Storage

VOLATILE DRAM

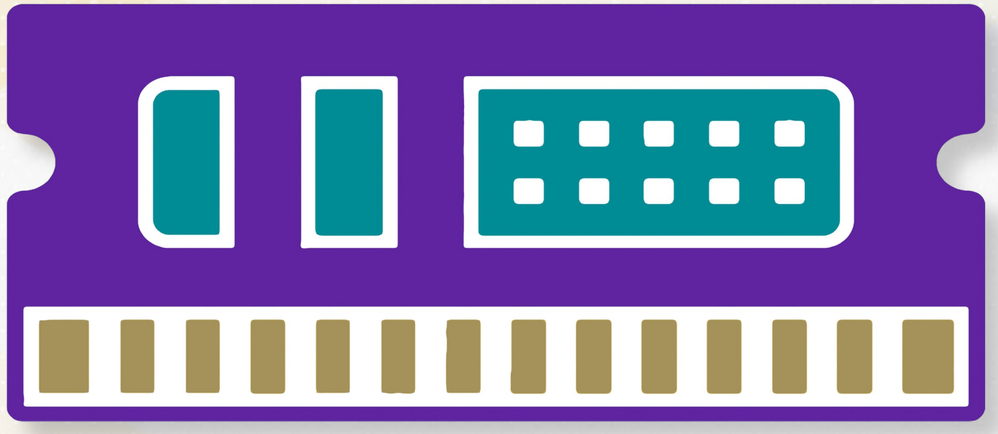


DRAM
(Dynamic Random-Access Memory)

Ideal for Quick Data Access

Data Lost Without Power

Persistent Storage



A Atomicity

C Consistency

I Isolation

D Durability

File I/O in C++

buzzDB


```
int main() {
  BuzzDB db;
  // Import data from “output.txt” file
  std::ifstream inputFile("output.txt");
  // Attempt to open file
  int field1, field2;
  // Read pairs of integers from the file and insert them into
BuzzDB
  while (inputFile >> field1 >> field2) {
    db.insert(field1, field2);
  }
  // Perform aggregation query on the imported data
  db.selectGroupBySum();
}
```

Tuple Class

Integer Key-Value Pairs	Strings

```
class Tuple {
public:
   // Identifier field
    int key;
    // Actual data field
    int value;
};
```

Field Class

Field Class

Type

FieldType

INT & FLOAT

Enums

Defines Constants

```
enum FieldType {INT, FLOAT}

class Field {
    public:
    FieldType type;
    union {
        int i;
        float f;
    } data;
};
```

Field Class

Integer Value INT

Float Value FLOAT

Constructor Overloading

Print Method

```
class Field {
    public:
    Field(int i): type(INT), data{i} {}
    Field(float f): type(FLOAT), data{f} {}
    void print() const {
        switch(type) {
            case INT: std:: cout << data.i;
            break;
            case FLOAT: std:: cout << data.f;
            break;
        }
    }
}
```

Generalized Tuple Class

```
class Tuple {
    std::vector<Field> fields;
    public:
    void addField(const Field & field) {
        fields.push_back(field);
    }
    void print() const {
        for (const auto & field: fields) {
            field.print(); std:: cout << " ";
        }
    }
};
```

Constructing Generalized Tuples

```
void BuzzDB::insert(int key, int value) {
  Tuple newTuple;
  Field key_field = Field(key);
  Field value_field = Field(value);
  float float_val = 132.04;
  Field float_field = Field(float_val);

  newTuple.addField(key_field);
  newTuple.addField(value_field);
  newTuple.addField(float_field);

  table.push_back(newTuple);
  index[key].push_back(value);
}
```

Further Generalization to Strings

FieldType enum includes STRING

```
enum FieldType {INT, FLOAT, STRING}

class Field {
    public:
    FieldType type;
    union {
        int i;
        float f;
        char *s; // string
    } data;
    ...
}
```

More Generalization to Strings

Dynamic Memory Allocation

String Parameter S

Field Object to String

String S + 1

Memory data.s

```
Field(const std::string &s) : type(STRING) {
  data.s = new char[s.size() + 1];
  std::copy(s.begin(), s.end(), data.s);
  data.s[s.size()] = '\0'; // Null-terminate the string
}
```

Smart Pointer in Field

Transition from manual memory management to std::unique_ptr for string

```
class Field {
public:
    int data_i;
    float data_f;
    std::unique_ptr <char[]> data_s;
    size_t data_s_length;
    ...
}
```

Smart Pointer in Field

unique_ptr

String Management Leak Risk Minimized

Destructor Eliminates delete[] Calls

```
Field(const std::string &s) : type(STRING) {
  data_s_length = s.size() + 1;
  data_s = std::make_unique<char[]>(data_s_length);
  std::strcpy(data_s.get(), s.c_str());
}
```

Smart Tuple Class

Tuple Class

std::vector

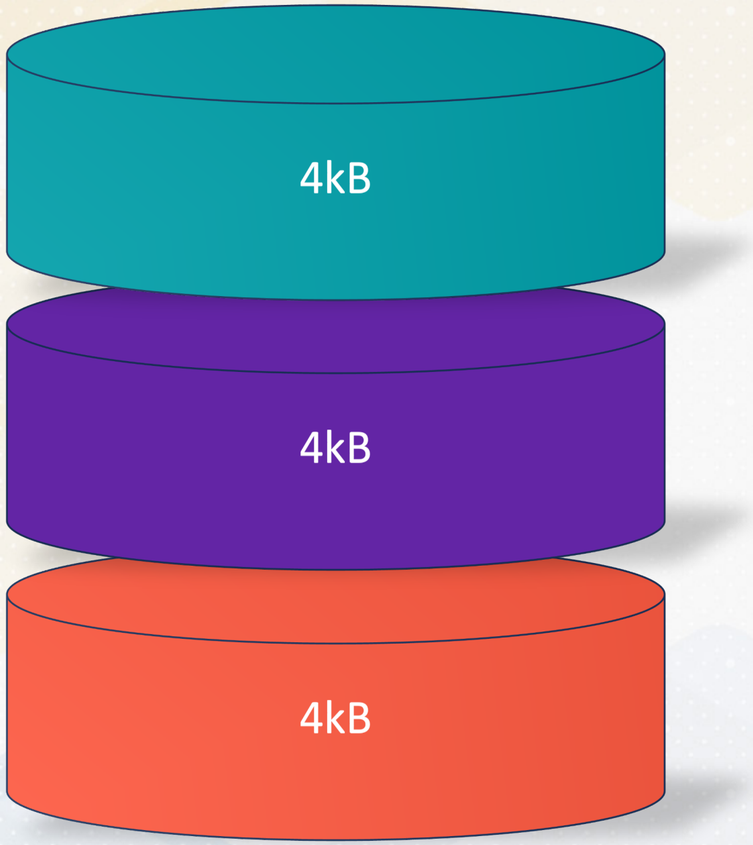
std::unique_ptr<Field>

```
class Tuple {
    std::vector< std::unique_ptr < Field >> fields;


    public:
    void addField(std::unique_ptr< Field > field) {
        fields.push_back(std:: move(field))
    }
};
```

unique_ptr cannot be copied in the traditional sense to ensure unique ownership

Tuples and Fields

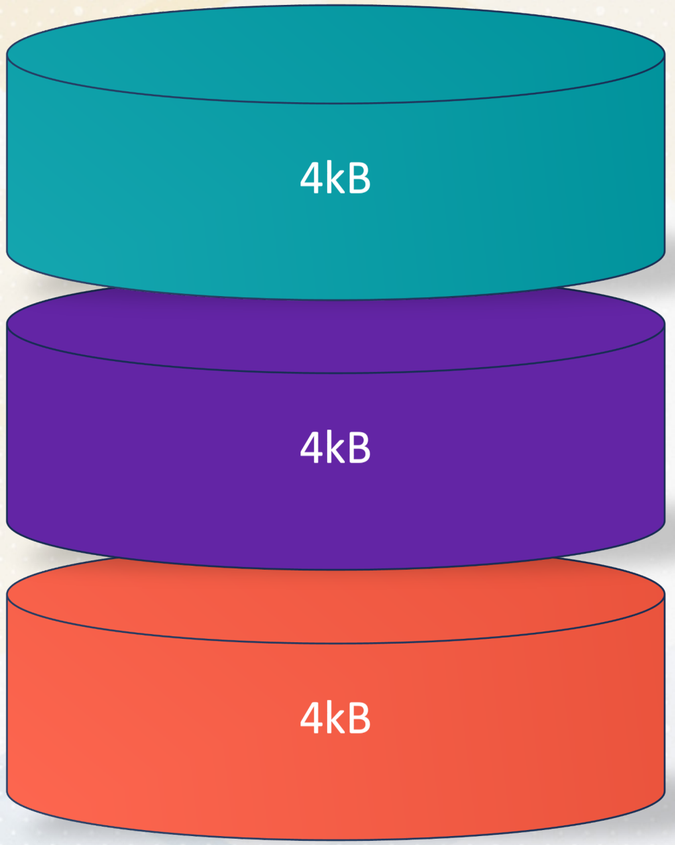
Disks and Pages



Basic unit of disk storage

```
class Page {
public:
  size_t used_size = 0;
  std::vector<std::unique_ptr<Tuple>> tuples;
};
```

Page Class



Dynamic Tuple Management

Maintains Page Capacity

```
bool addTuple(std::unique_ptr<Tuple> tuple) {
  size_t tuple_size = // Calculated size;
  if (used_size + tuple_size > PAGE_SIZE)
        return false;
  tuples.push_back(std::move(tuple));
  used_size += tuple_size;
  return true;
}
```

Serialization of Page

Page Class Serialization Process

Write the number of tuples in Page

Loop through each tuple and serialize its data

```
void write(const std::string &filename) const {
  std::ofstream out(filename);
  // First write the number of tuples.
  size_t numTuples = tuples.size();
  out.write(reinterpret_cast<const char *>(&numTuples), sizeof(numTuples));
  ...
}
```

Serialization of Tuple

Page Class Serialization Process

Write the number of tuples in Page

Loop through each tuple and serialize its data

Record how many fields each tuple contains

Field serialization involves: *Field Type*, *Field Length*, and *Field Data*

Serialization of Field

```
// Then write each tuple.
for (const auto &tuple : tuples) {
  // Write the number of fields in the tuple.
  size_t numFields = tuple->fields.size();
  out.write(reinterpret_cast<const char *>(&numFields), sizeof(numFields));

  // Then write each field.
  for (const auto &field : tuple->fields) {
    out.write(reinterpret_cast<const char *>(&field->type), sizeof(field->type));
    out.write(reinterpret_cast<const char *>(&field->data_length), sizeof(..));
    out.write(field->data.get(), field->data_length);
  }
}
```

Deserialization of Page

Deserialization allows BuzzDB to load pages stored on disk

Deserialization reconstructs in-memory pages

```
// Read this page from a file.
void read(const std::string &filename) {
  std::ifstream in(filename);
  // First read the number of tuples.
  size_t numTuples;
  in.read(reinterpret_cast<char *>(&numTuples), sizeof(numTuples));
}
```

Deserialization of Tuple

Identify the number of fields in a tuple

```
// Read the number of fields in the tuple.
size_t numFields;
in.read(reinterpret_cast<char *>(&numFields), sizeof(numFields));


// Then read each tuple.
for (size_t i = 0; i < numTuples; ++i) {
  auto tuple = std::make_unique<Tuple>();
  // Read the number of fields in the tuple.
  size_t numFields;
  in.read(reinterpret_cast<char *>(&numFields), sizeof(numFields));
```

Deserialization of Field

- Reading the field type and its length
- Reading the data
- Reconstructing the field in memory

```
// Then read each field.
for (size_t j = 0; j < numFields; ++j) {
    // Read the type of the field.
    FieldType type;
    in.read(reinterpret_cast<char *>(&type), sizeof(type));
    // Read the length of the field.
    size_t data_length;
    in.read(reinterpret_cast<char *>(&data_length), sizeof(data_length));
    // Then read the field data.
    std::unique_ptr<char[]> data(new char[data_length]);
    in.read(data.get(), data_length);
```

Deserialization

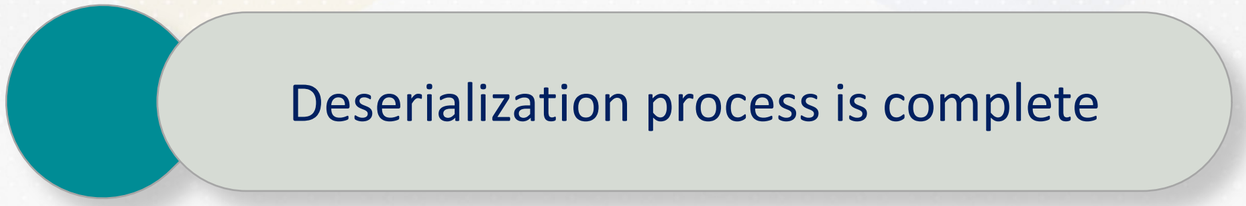
Tuple Restoration

fields are assembled back into their respective tuples based on their type, whether integer, float, or string

```
// Add the field to the tuple.
   switch (type) {
     ...
     case STRING: {
         char *val = reinterpret_cast<char *>(data.get());
         auto field = std::make_unique<Field>(std::string(val, data_length));
         tuple->addField(std::move(field));
         break;
     }
 }
```

Deserialization

Tuple Restoration




```
std::cout << "Tuple " << (i + 1) << " :: ";
tuple->print();
```

Using Page Class in BuzzDB

Tuples pushed to a Page

```
class BuzzDB {
    Page page;
    void insert(int key, int value) {
        page.addTuple(std::move(newTuple));
    }
};
int main() {
    // Serialize page to disk
    db.page.write(filename);
    // Deserialize page from disk
    Page page2;
    page2.read(filename);
}
```

Data Inconsistency

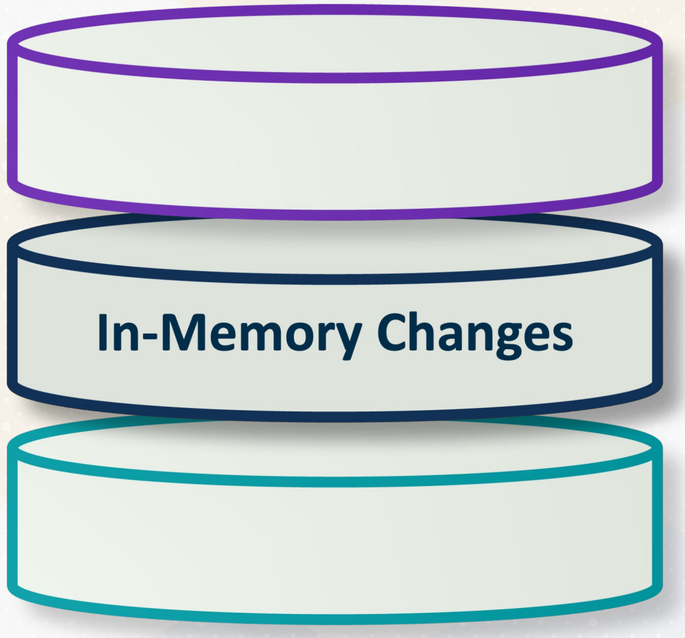
In-Memory Deletion

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);
```

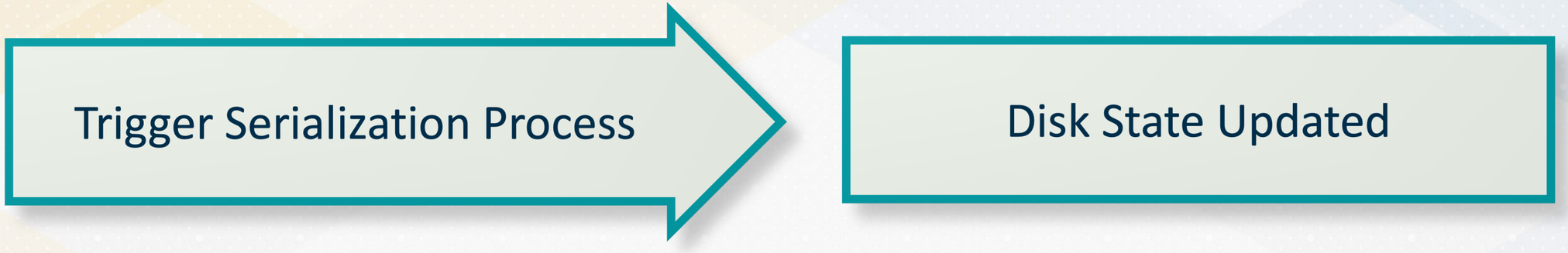
Data Inconsistency



Data Inconsistency Issues

- In-Memory Changes Not Persisted
- Memory and Disk State Discrepancy

Data Synchronization

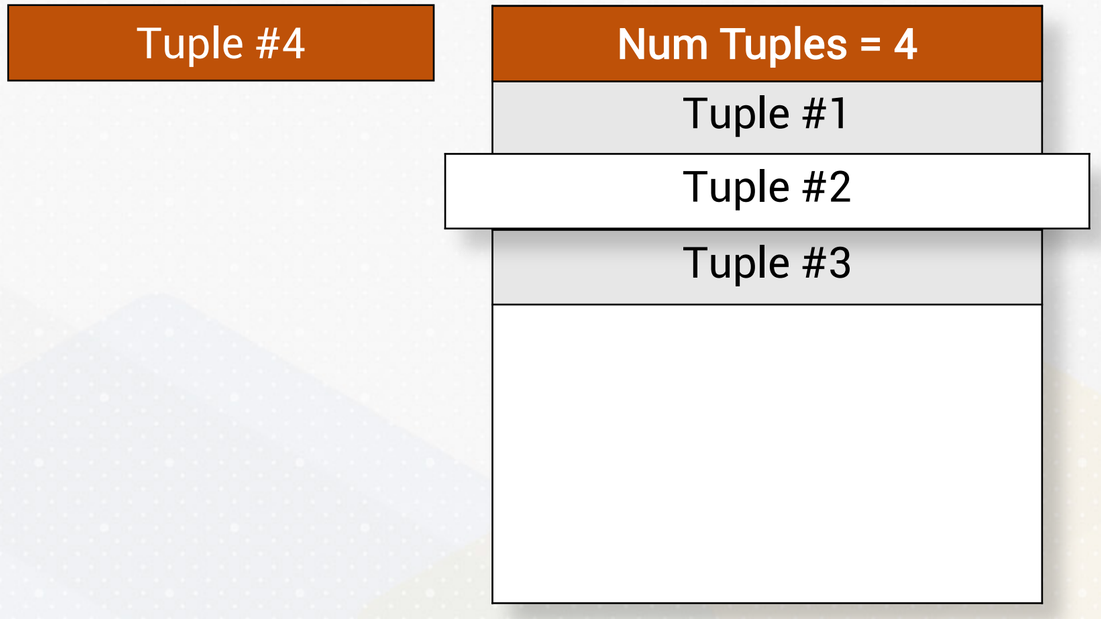



```
std::cout << "Deleting slots 0 and 7 \n";
loadedPage->deleteTuple(0);
loadedPage->deleteTuple(7);

loadedPage->write(filename);
```

Slotted Page

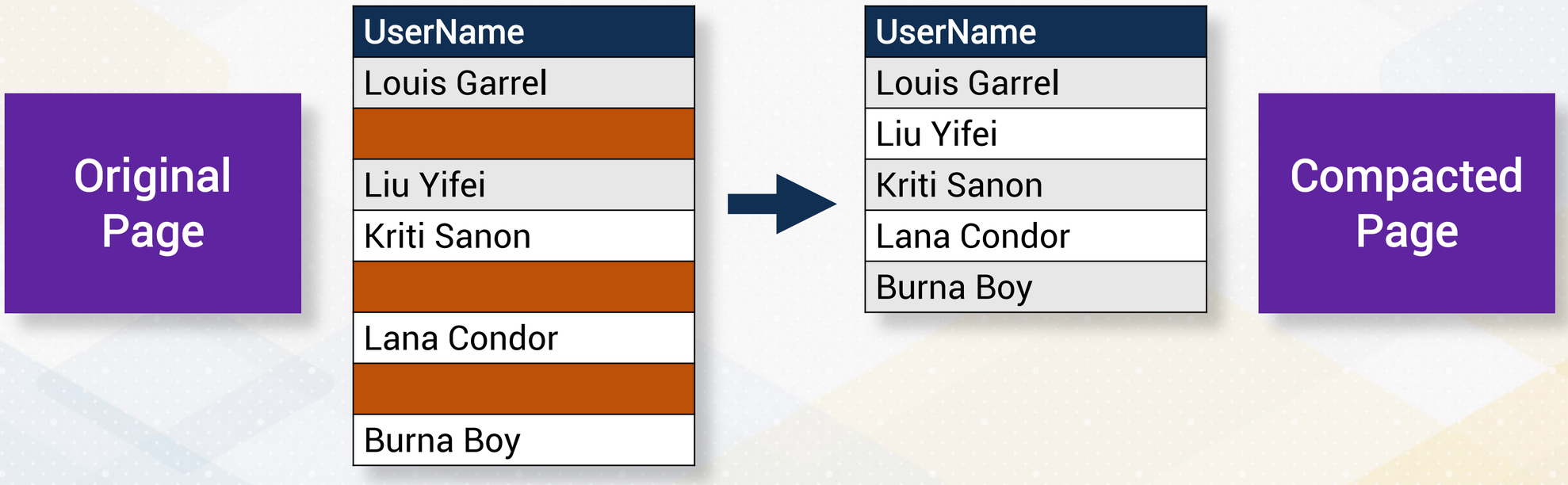
Limitations of Page Class: Linear Scan

- 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.



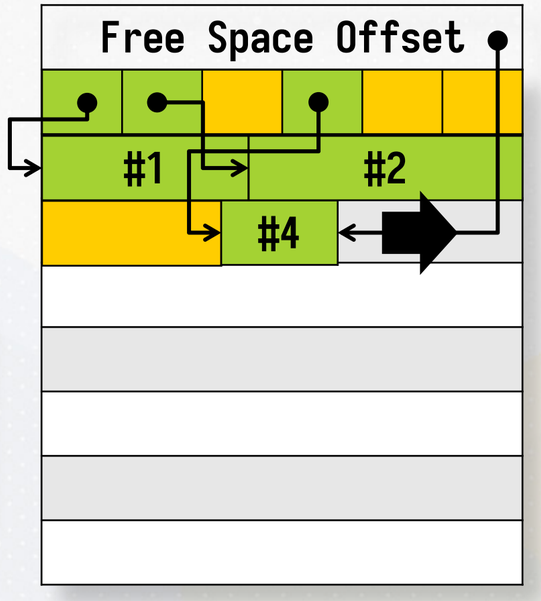
Slotted Page

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

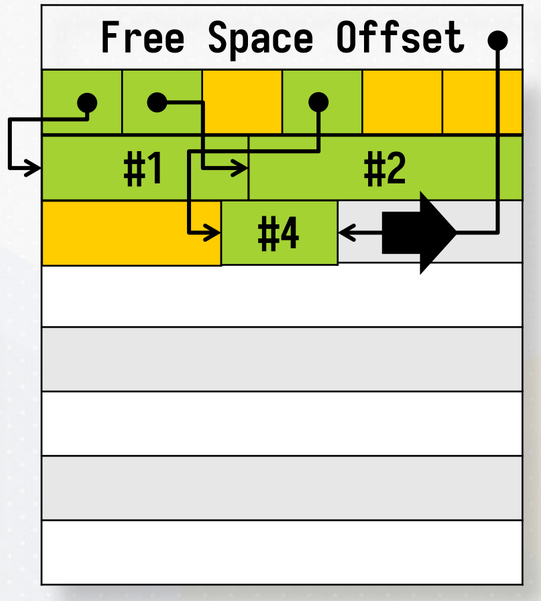
Slotted Page: Deleting Tuple

```
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.



Slot Array

Variable Length Tuples

Database File Management

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

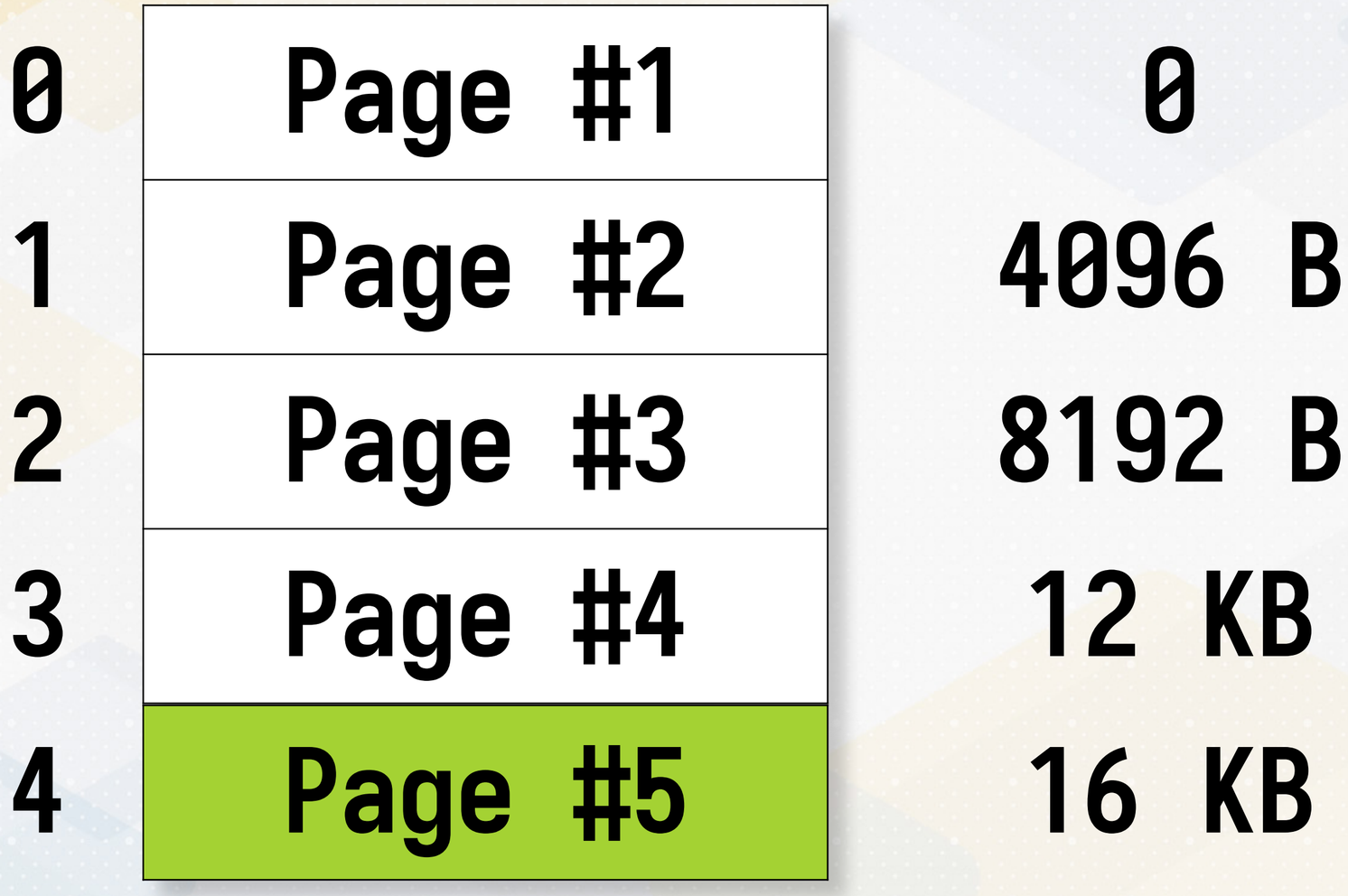
```
class BuzzDB {
private:
  std::fstream file;
  // a vector of Slotted Pages acting as a table
  std::vector<std::unique_ptr<SlottedPage>> pages;

public:
  BuzzDB() { file.open(database_filename, std::ios::in | std::ios::out); }

};
```

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



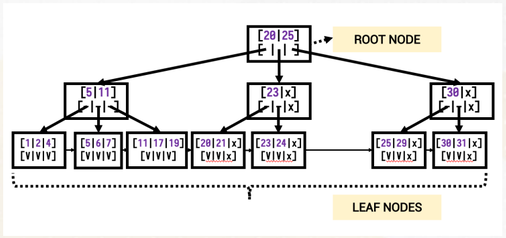
What is an Index?

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



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
    }
  }
}
```

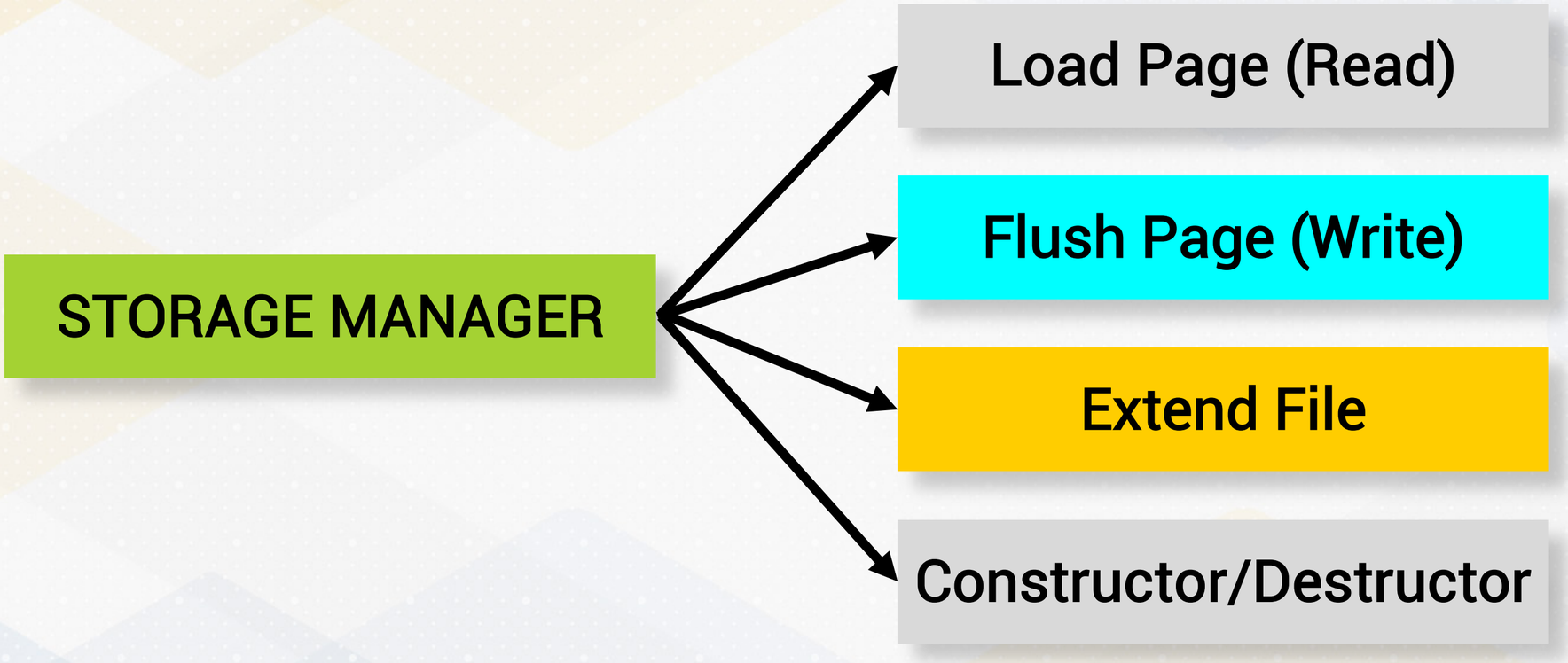
Index Construction

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
  }
}
```

Storage Manager

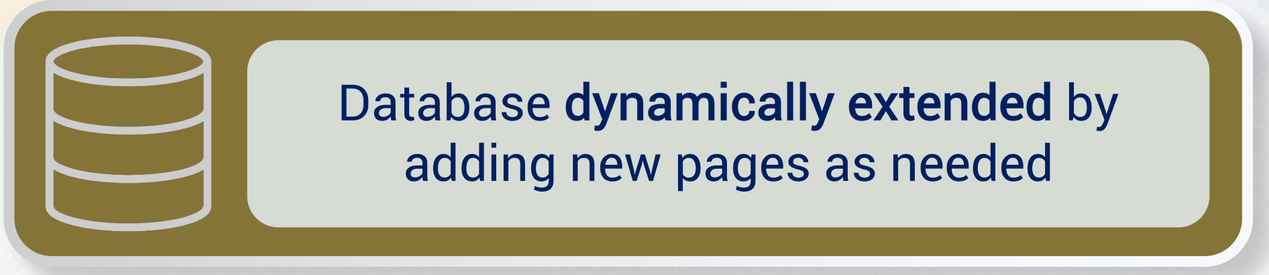
Storage Manager



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

flush operation

- Ensures changes made to in-memory pages are written back to the disk
- Secure data against potential data loss

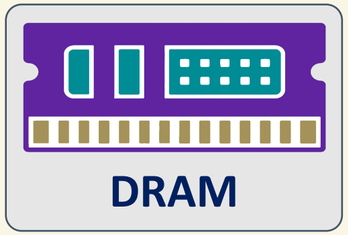
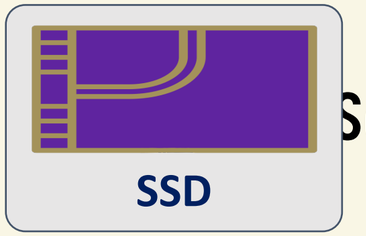
```
void StorageManager::flush(uint16_t page_id) {
  size_t page_offset = page_id * PAGE_SIZE;
  fileStream.seekp(page_offset, std::ios::beg);
  fileStream.write(pages[page_id]->page_data.get(), PAGE_SIZE);
  fileStream.flush();
}
```

Data Loading


```
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;
}
```

Buffer Manager

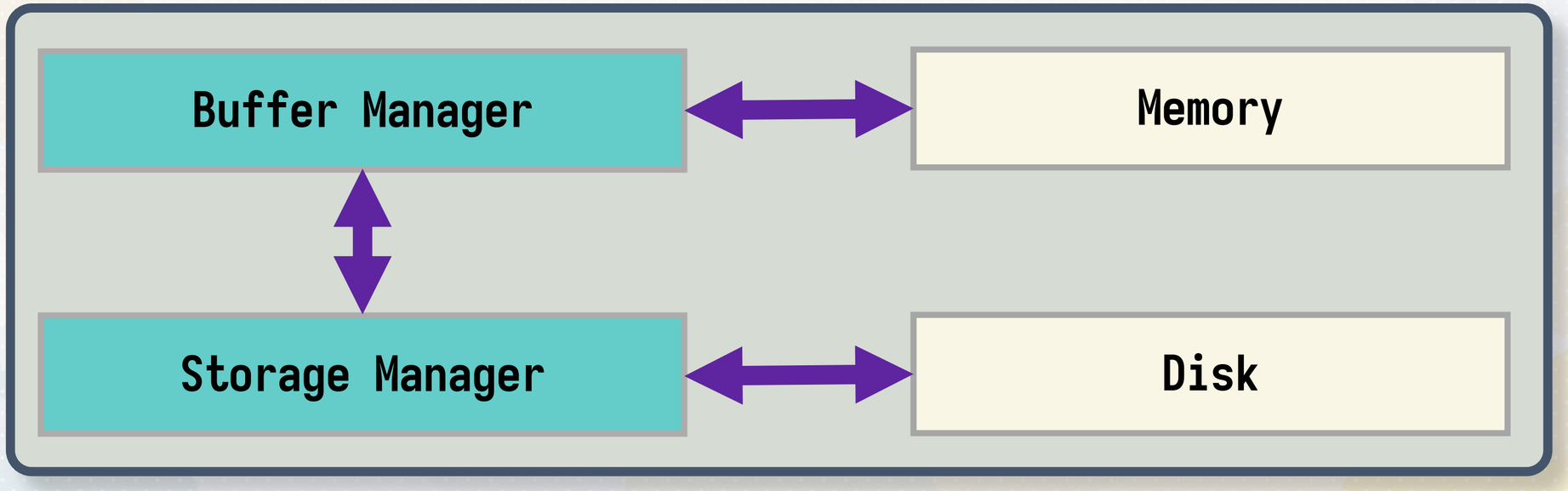
DRAM vs SSD

Device	Latency (ns)	Price per GB ($)
 Dynamic RAM (DRAM)	50	10
 Solid State Disk (SSD)	50 * 1000	0.1

Buffer Manager

Buffer Manager Caches frequently-accessed pages in DRAM

Storage Manager: low-level file I/O operations

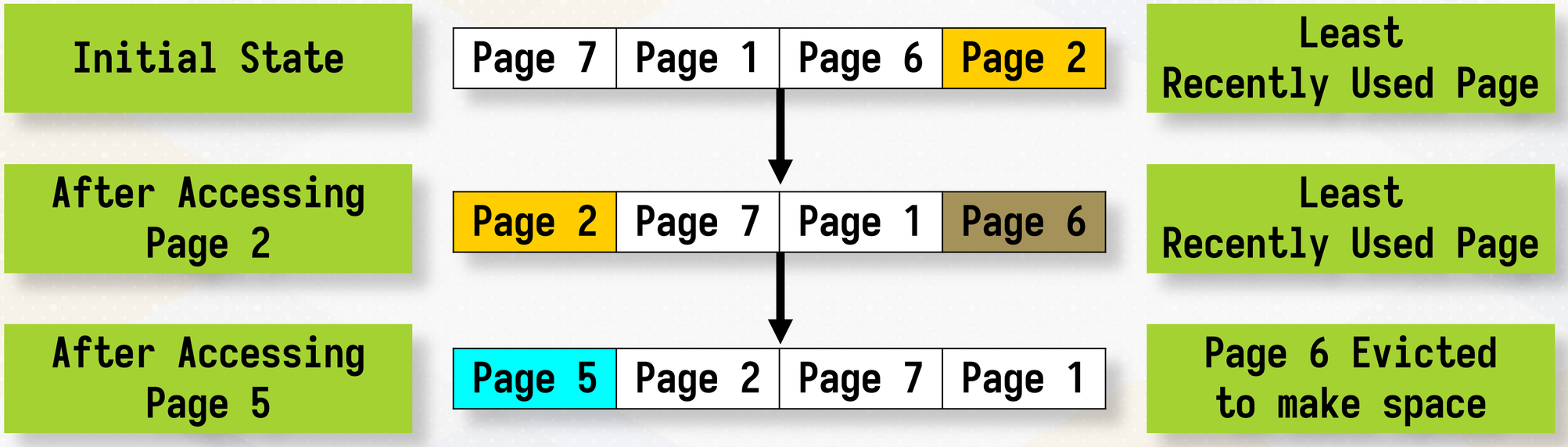


Buffer Manager

Buffer Manager: Maintains a **list of pages** in memory and a mapping from page IDs

```
class BufferManager {
private:
  StorageManager storage_manager;
  // LRU list to maintain the order of page usage
  std::list<std::pair<PageID, std::unique_ptr<SlottedPage>>> lruList;
  // Map to quickly find pages in the list
  std::unordered_map<PageID, typename PageList::iterator> pageMap;
public:
  // Fetches a page with given ID, applying LRU policy for caching
  std::unique_ptr<SlottedPage> &getPage(int page_id);
};
```

Least Recently Used (LRU) Policy



Buffer Manager

PageID

Utilized for page identification

lruList

Stores currently loaded pages in LRU order

pageMap

mapping from PageID to their locations in lruList

```
class BufferManager {
private:
  StorageManager storage_manager;
  // LRU list to maintain the order of page usage
  std::list<std::pair<PageID, std::unique_ptr<SlottedPage>>> lruList;
  // Map to quickly find pages in the list
  std::unordered_map<PageID, typename PageList::iterator> pageMap;
};
```

Buffer Manager & Storage Manager

StorageManager: Loads Page from Disk when Page Not Found in Buffer Cache

```
std::unique_ptr<SlottedPage> &BufferManager::getPage(int page_id) {
  auto it = pageMap.find(page_id);
  if (it == pageMap.end()) {
    // Page not in cache, load from disk
    auto page = storage_manager.load(page_id);
    ...
  }
  touch(pageMap[page_id]);
  return lruList.begin()->second;
}
```

touch

touch

- Updates a page's position in the **LRU list** to reflect its recent access
- Splice method in **std::list** is used to move the page to the front of the list

```
void touch(PageList::iterator it) {
  // Move page to the front of the list, indicating recent use
  lruList.splice(lruList.begin(), lruList, it);
}
```

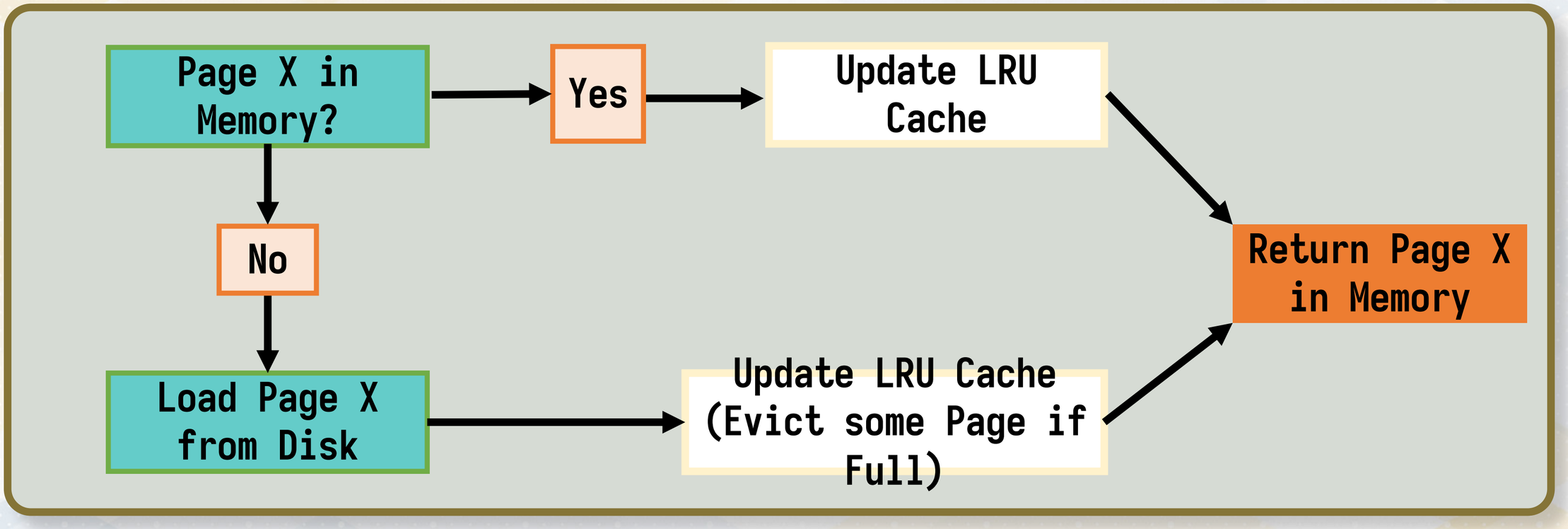
evict

evict

- Removes least recently used page from buffer
- Changes flushed to ensure no data loss

```
void evict() {
  // Get iterator to the last element
  auto last = --lruList.end();
  int evictedPageId = last->first;
  // Flush page changes to disk before eviction if necessary
  storage_manager.flush(evictedPageId, last->second);
  // Remove page from page map and LRU list
  pageMap.erase(evictedPageId);
  lruList.pop_back();
}
```

Buffer Manager



Modularity

buzzDB

- Interacts only with **BufferManager** for page-related operations
- BuzzDB no longer accesses **StorageManager**

```
// BuzzDB now uses BufferManager for page operations
void BuzzDB::insert(int key, int value) {
  ...
  // Interacts through BufferManager
  auto &page = buffer_manager.getPage(page_id);
  page->addTuple(std::move(newTuple));
  ...
}
```

Sequential Scan

sequential scan	when a database reads every table page while processing a query

```
// Sequential scan across the sales_data table
// Assumption: No index on sale_date column
SELECT *
FROM sales_data
WHERE sale_date BETWEEN '2040-01-01' AND '2040-01-31';
```

Buffer Pool Flooding

Sequential scan floods the buffer pool with less important pages

LRU Policy

Evicts hot pages to store cold pages

LRU: Buffer Pool Flooding

Page Access Trace

1	2	1	2	3	4	1	2	5	6	1	2

Buffer Pool Capacity: 3 Pages

LRU evicts **hot** pages 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
LRU Queue: 1	1	2	1	2	3	4	1	2	5	6	1	2
LRU Queue: 2	-	1	2	1	2	3	4	1	2	5	6	1
LRU Queue: 3	-	-	-	-	1	2	3	4	1	2	5	6
Evict Page						1	2	3	4	1	2	5
Cache Misses	1	2	2	2	3	4	5	6	7	8	9	10

TwoQ Policy

page transition logic	• Page initially enters **FIFO** queue • Page moves to **LRU** queue on second access

FIFO Queue

Read-Once Pages Accessed During Sequential Scan

LRU Queue

Frequently-Accessed **Hot** Pages Promoted to LRU Queue

TwoQ Policy: touch

Pages Enter **FIFO** Queue

Pages Promoted To **LRU**

Page Position Updated When Accessed

```
void touch(PageID page_id) {
  if (pageMap.find(page_id) != pageMap.end()) {
    auto it = pageMap[page_id];
    // If in FIFO and accessed again, move to LRU
    // Otherwise, adjust position within LRU
    // If cache is full, then evict
    // New pages added to FIFO
  }
}
```

TwoQ: No Buffer Pool Flooding

TwoQ retains **hot** pages like 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
FIFO Queue:1	1	2	2	_	3	4	4	4	5	6	6	6
FIFO Queue:2	-	1	-	-	-	-	-	-	-	-	-	-
FIFO Queue:3	-	-	-	-	-	-	-	-	-	-	-	-
LRU Queue: 1	-	-	1	2	2	2	1	2	2	2	1	2
LRU Queue: 2	-	-	-	1	1	1	2	1	1	1	2	1
LRU Queue: 3	-	-	-	-	-	-	-	-	-	-	-	-
Evict Page						3			4	5		
Cache Misses	1	2	2	2	3	4	4	4	5	6	6	6

Conclusion

- Storage Management
- Smart Pointers and Pages
- Slotted Page
- Buffer Management
- 2Q Policy