



Lecture 10: Database Configuration



Logistics

- Point Solutions App
 - Session ID: **database**
- Programming assignment 2 due on **Oct 1** (Gradescope)
- Practice questions for exercise sheet 1 (Honorlock + Canvas)

Recap

- Multi-Threading
- Synchronization
- Fine-Grained Locking
- Debugging

Lecture Overview

- Database Configuration
- Indexing in C++
- Hash Table
- Hash Function
- Key Deletion

Database Configuration

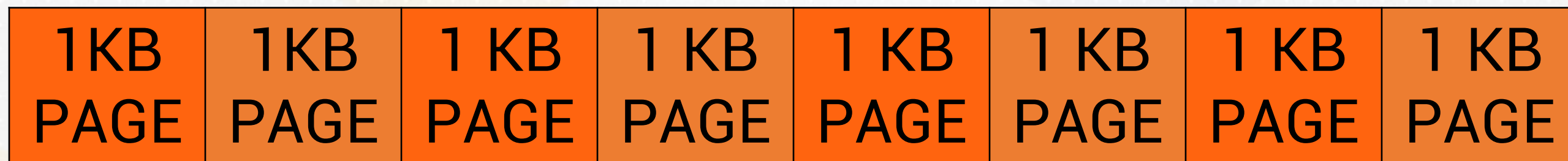


Database Configuration

Parameter	Value
PAGE_SIZE	1 KB
MAX_SLOTS	128 slots per page
MAX_PAGES_IN_MEMORY	10 pages
...	...

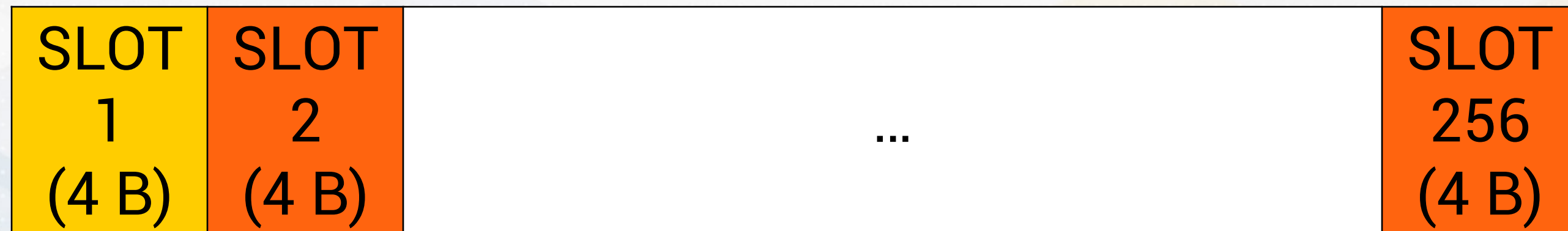
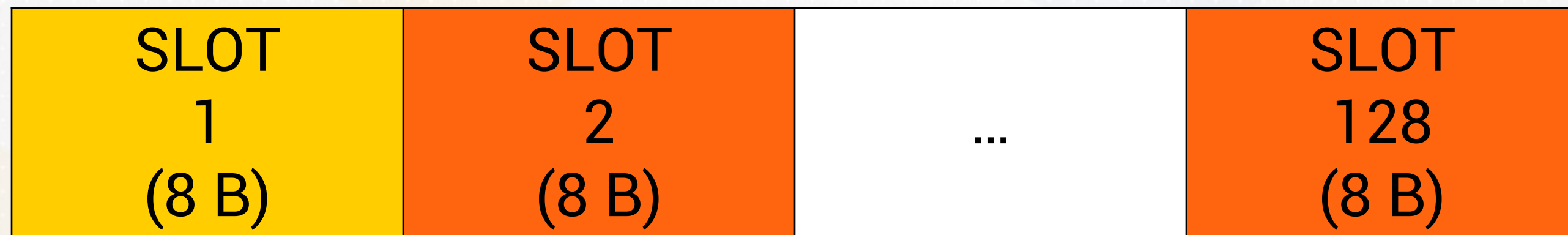
PAGE_SIZE

8KB BUFFER POOL



MAX_SLOTS

1KB SLOTTED PAGE



MAX_PAGES_IN_MEMORY

BUFFER POOL

PAGE 1	PAGE 2	PAGE 3	PAGE 4
-----------	-----------	-----------	-----------

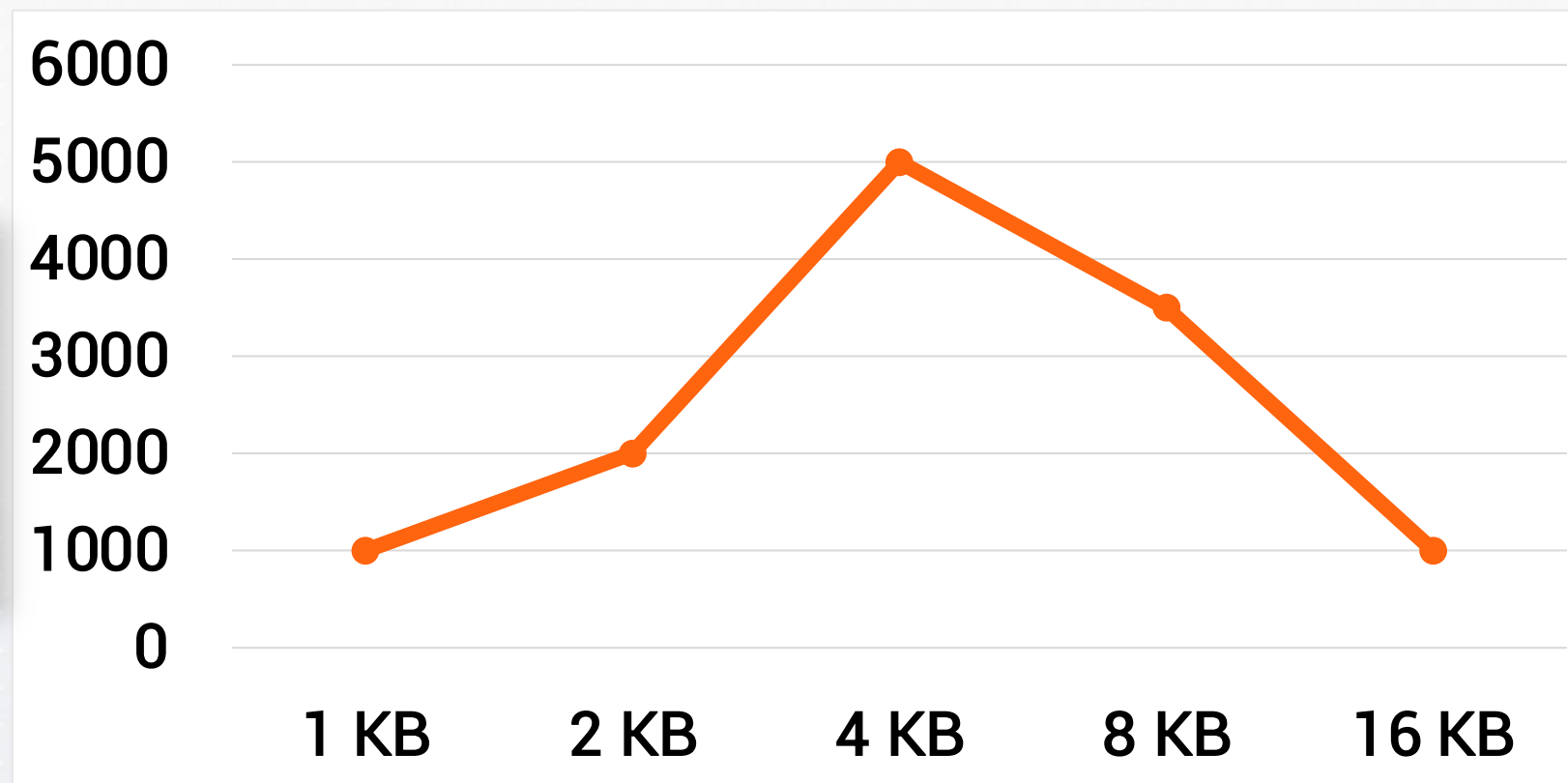


PAGE 1	PAGE 2	PAGE 3	PAGE 4	PAGE 5	PAGE 6	PAGE 7	PAGE 8
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Challenges in Configuration Tuning

Most parameters like `PAGE_SIZE` have a “sweet spot”

THROUGHPUT
(TRANSACTIONS
PER SECOND)



PAGE SIZE PARAMETER

Challenges in Configuration Tuning

PostgreSQL's default configuration has low buffer pool size

System Parameter	Value
BUFFER_POOL_SIZE	128 MB



System Parameter	Value
BUFFER_POOL_SIZE	16 GB

Immutable vs. Mutable Parameters

System Parameter	Mutable or Immutable
PAGE_SIZE	Immutable
BUFFER_POOL_SIZE	Mutable

Cloud Database Tuning

Cost vs Performance

Memory Size	Throughput	Cost	Price-Performance
8 GB	100 transactions per second	\$0.05 per hour	\$0.139 per million transactions
16 GB	200 transactions per second	\$0.15 per hour	\$0.208 per million transactions

Tools for Configuration Tuning

Automatically recommend configuration
based on hardware and application

Parameter	Value
TOTAL_MEMORY	16 GB
NUMBER_OF_CPUS	16
STORAGE_TYPE	SSD / HDD / SAN
APPLICATION_TYPE	Type 1 / Type 2 / Type 3

Database Application Type

Application Type	Performance Bound by	Database Size Relative to DRAM	Workload Characteristics
Online Transaction Processing (OLTP)	CPU- or I/O-bound	DB slightly larger than DRAM ~ 1 TB	20-40% short transactions, some long transactions and queries
Online Analytical Processing (OLAP)	I/O- or DRAM-bound	DB much larger than DRAM ~ 4 TB	Large complex reporting queries; Also known as data warehouse
Web Application (WEB)	CPU-bound	DB much smaller than DRAM ~ 4 GB	90% or more simple queries

Reinforcement Learning for Configuration Tuning

- Agent interacts with database by trying different configuration settings
- Observes the impact on performance such as throughput

State	Current settings of database parameters and the current workload characteristics
Action	Increase or decrease the value of some parameter
Reward	Increase in throughput or decrease in price-performance metric

OS Support for DB



OS Support for Database Management

COMPUTING PRACTICES

Operating System Support for Database Management

Michael Stonebraker
University of California, Berkeley

1. Introduction

Database management systems (DBMS) provide higher level user support than conventional operating systems. The DBMS designer must work in the context of the OS he/she is faced with. Different operating systems are designed for different use. In this paper we examine several popular operating system services and indicate whether they are appropriate for support of database management functions. Often we will see that the wrong service is provided or that severe performance problems exist. When possible, we offer some

SUMMARY: Several operating system services are examined with a view toward their applicability to support of database management functions. These services include buffer pool management; the file system; scheduling, process management, and interprocess communication; and consistency control.

suggestions concerning improvements. In the next several sections we look at the services provided by buffer pool management; the file system; scheduling, process management, and interprocess communication;

the operating system is compiled. Then, all file I/O is handled through this cache. A file read (e.g., read X in Figure 1) returns data directly from a block in the cache, if possible; otherwise, it causes a block to be

Buffer Management: Policy

- OS buffer pools often use LRU replacement policy.
- For DB workloads LRU is often suboptimal.
- Access patterns in INGRES are four types:
 - Sequential access of blocks *not* rereferenced
 - Sequential access of blocks which *will* be cyclically rereferenced
 - Random access to blocks that won't be used again
 - Random access where there's some chance of reaccess.
- LRU only works well in case 4, but badly in other cases.

Buffer Management: Prefetching

- OSes like UNIX detect sequential access and prefetch blocks
- DBMS often knows in advance which block(s) will be needed next, and the next needed block is not always “next in file order” logically.
- So, OS prefetching is frequently inadequate or mis-aligned.

File System: File Abstraction

- UNIX provides files as character arrays
- DBMS builds on top of that whatever higher-level objects it needs (records, indexing, etc.).
- UNIX extends files one block at a time, so blocks end up scattered across disk.
- For sequential scans this creates too much disk arm motion.
- DB prefers “extent”-based storage to reduce seek overhead.

File System: Indexing

- UNIX uses indirect blocks (tree of blocks) for file contents; directory structure is a tree too.
- DBMS adds its own B-trees for indexing. So there are multiple tree structures layered, often redundantly.

Indexing in C++



Indexing in Database Systems

INDEX	
Klein, Arno	249
La Duke, Elliott W.	377
Lawless, Frank R.	307
Layer, John G.	330
Leffel, John C.	357
Leffel, Edward	359
Leonard, Frederick P.	371
Lewis, Edward	338
Lewis, Frank E.	290
Lewis, James R.	327
Lewis, Thompson P.	339
Llewelyn, Edgar J.	331
Llewelyn, George T.	334
Llewellyn, Francis B.	298
Llewellyn, Enoch W.	387
Llewellyn, James F.	389
Llewellyn, Noah	389
McReynolds, Samuel M.	296
MacGregor, Francis B.	244
Macy, Carlos B.	287
Marvel, Alexander L.	400
Marvel, Thomas	397
Meinschein, Conrad	338
Menzies, G. V.	217
Menzies, Winston	218
Miller, Lorenz C.	385
Moeller, John H.	256
Montgomery, Samuel B.	282
Morrow, Lannie G.	350
Moye, James H.	362

Book Index

Index Finger

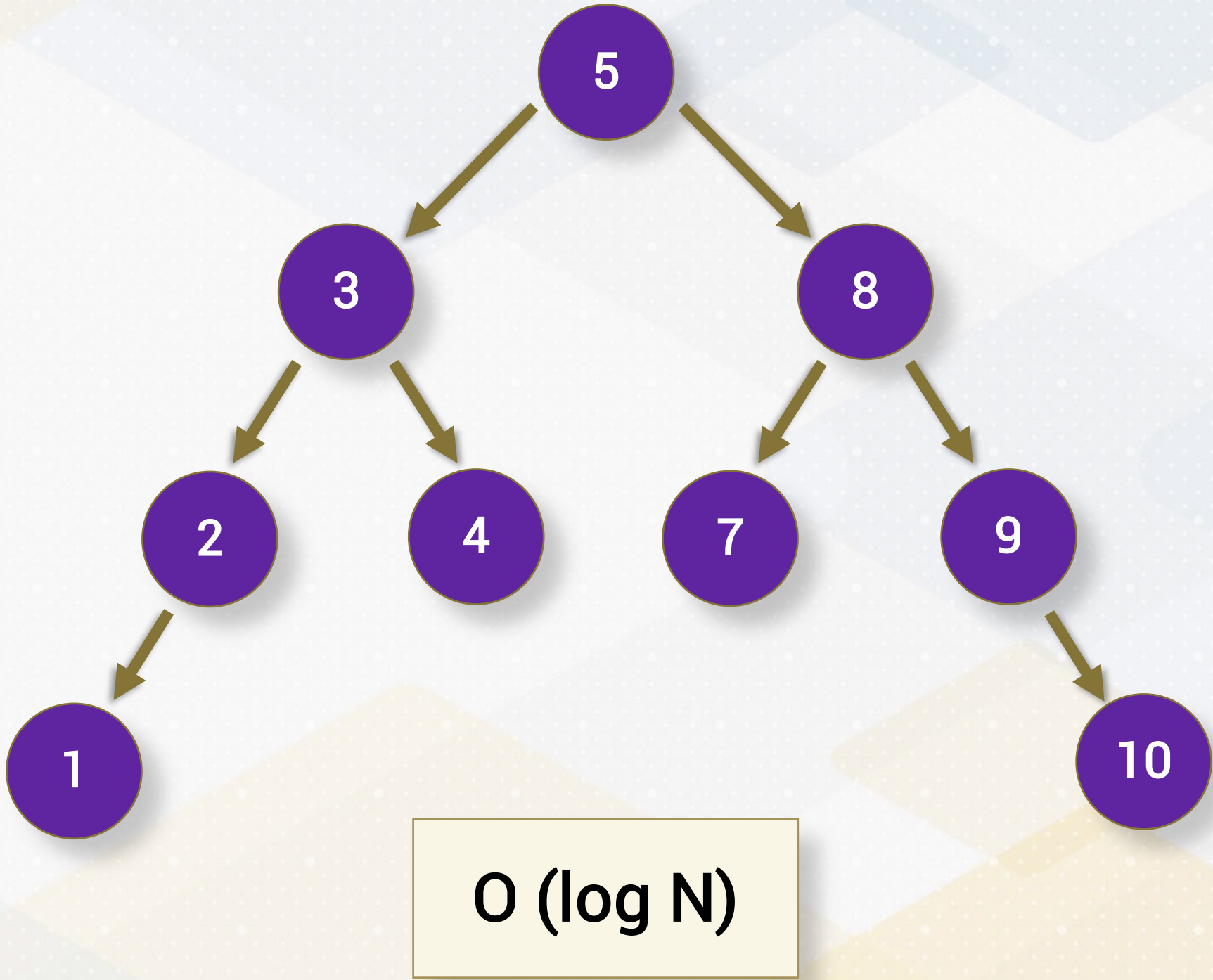


Indiana State Library and Historical Bureau, Public domain, via Wikimedia Commons



Indexing in C++: ordered_map

Key	Value
1	Apple
2	Banana
3	Cherry
4	Date
5	Elderberry
6	Fig
7	Grape
8	Honeydew
9	Ilama
10	Jackfruit



Indexing in C++: unordered_map

Key	Value
2	Banana
6	Fig
1	Apple
3	Cherry
4	Date
5	Elderberry
8	Honeydew
7	Grape
10	Jackfruit
9	Ilama

Avg Case

$O(1)$

Worst Case

$O(N)$

map vs unordered_map

	MAP	UNORDERED MAP
Ordering	Sorted based on key	Not sorted based on key
Implementation	Binary Search Tree	Hash Table
Average Case Time	$O(\log N)$	$O(1)$
Worst Case Time	$O(\log N)$	$O(N)$

HashIndex in BuzzDB

insertOrUpdate

getValue

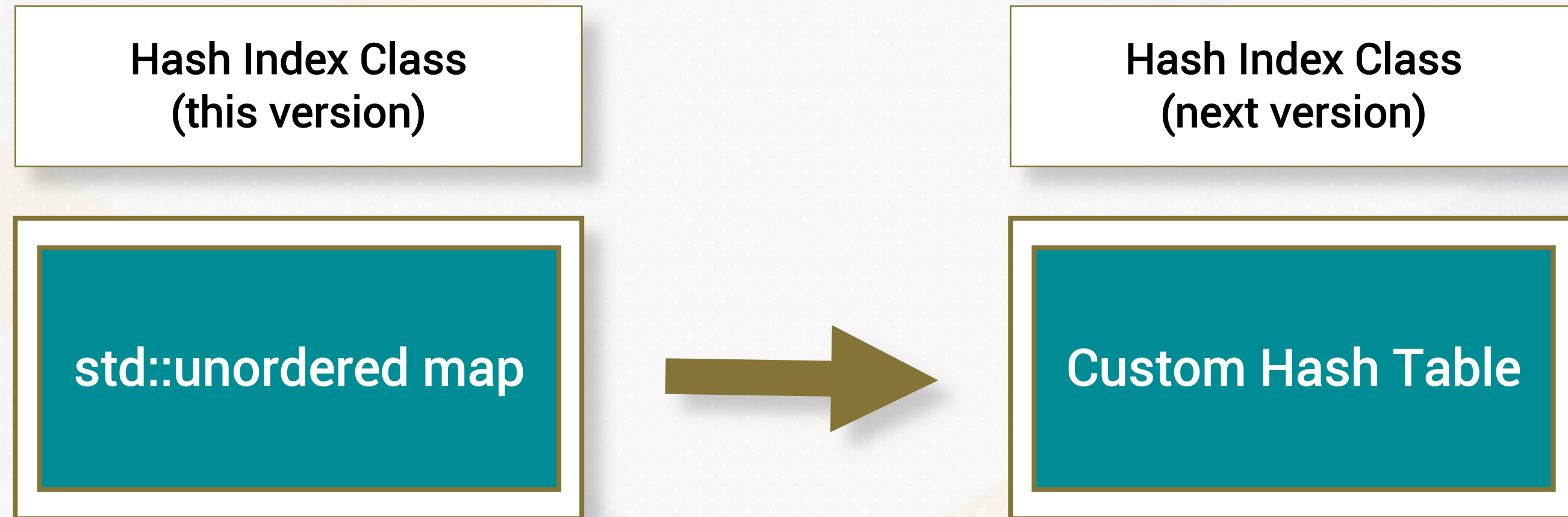
```
class HashIndex {  
private:  
    std::unordered_map<int, int> hash_index;  
public:  
    void insertOrUpdate(int key, int value) {  
        hash_index[key] = value;  
    }  
    int getValue(int key) const {  
        auto it = hash_index.find(key);  
        return it != hash_index.end() ? it->second : -1;  
    }  
};
```


Integration of HashIndex in BuzzDB

During the scanning phase, each tuple's key-value pair is inserted into the HashIndex.

```
void BuzzDB::scanTableToBuildIndex() {  
    // Iterating over pages and tuples  
    for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {  
        // Extract key-value pairs and insert them into the index  
        int key = loadedTuple->fields[0]->asInt();  
        int value = loadedTuple->fields[1]->asInt();  
  
        index.insertOrUpdate(key, value);  
    }  
}
```

Custom HashTable



Hash Table



Hash Table

Think of a hash table like a cabinet of drawers.
Each drawer can hold a piece of paper with the **key** and **associated value**.

0	1	2	3	4	5	6	7	8	9
					15				
					Apple				



DIRECT ACCESS USING HASH FUNCTION

Inserting Key-Value Pair

Insert (15, Apple)

0	1	2	3	4	5	6	7	8	9
					15				
					Apple				



$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 15 \% 10 = 5$ (since $15 = 10 * 1 + 5$)

Inserting Key-Value Pair

Insert (26, Grape)

0	1	2	3	4	5	6	7	8	9
					15	26			
					Apple	Grape			



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 26 \% 10 = 6 \text{ (since } 26 = 10 * 2 + 6 \text{)}$$

Inserting Key-Value Pair with Collision

Insert (56, Kiwi)

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 56 \% 10 = 6$$

Finding Key-Value Pair

Find value associated with key 15 = Apple

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 15 \% 10 = 5$$

Finding Key-Value Pair

Find value associated with key 56 = Kiwi

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 56 \% 10 = 6$$

Finding Key-Value Pair

Find value associated with key 86 = KEY NOT FOUND

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 86 \% 10 = 6$$

Hash Function



Hash Function

Hash function maps keys to slots in the hash table

```
size_t hashFunction(int key) {  
    return key % totalSlots; // Our simple formula  
}
```

$$\text{HASH}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

Collision Handling

Linear Probing

Move to the next available slot until we find one empty.

0	1	2	3	4	5	6	7	8	9
80	11				15	26	56	18	79
Lime	Date				Apple	Grape	Kiwi	Fig	Pear



If slot 8 is taken, check slot 9, then 0, 1, and so on.

"Linear" Probing

Formula to find the index of the next slot I after K "probes" is "linear" with respect to K.

$$\text{INDEX } I = (\text{HASH}(\text{KEY}) + K) \% \text{TABLE_SIZE}$$

$$\text{HASH}(\text{KEY}) = \text{KEY} \% \text{TABLE_SIZE}$$

HashEntry Struct

HashEntry Struct

Represents a piece of paper containing **key** and **value**.

```
struct HashEntry {  
    int key, value;  
    HashEntry(int k, int v) : key(k), value(v) {}  
};
```

HashIndex

HashTable is simply an array of drawers. Each drawer can either hold a `HashEntry` or be **empty** (indicating no value).

```
class HashIndex {  
    std::vector<std::optional<HashEntry>> hashTable;  
  
    static const size_t capacity = 100; // Hard-coded capacity  
    HashIndex() { hashTable.resize(capacity); }  
};
```


Inserting Key-Value Pair

insertOrUpdate

Uses linear probing to find the next available slot for insertion or locate an existing key for updating.

```
void insertOrUpdate(int key, int value) {
    size_t index = hashFunction(key);
    do {
        if (!hashTable[index]) {
            hashTable[index] = HashEntry{key, value, true}; // Insert new entry
            break;
        } else if (hashTable[index]->key == key) {
            hashTable[index]->value += value; // Update existing entry
            break;
        }
        index = (index + 1) % capacity; // Linear probing to next slot
    } while (index != originalIndex); // Returned to starting point
}
```

Finding Key-Value Pair

getValue

The retrieval process employs linear probing as well to navigate through potential collision sequences.

```
int getValue(int key) const {  
    size_t index = hashFunction(key);  
    do {  
        if (hashTable[index] && hashTable[index]->key == key) {  
            return hashTable[index]->value; // Key found  
        }  
        index = (index + 1) % capacity; // Continue probing  
    } while (index != originalIndex);  
    return -1; // Key not found  
}
```


Operation Complexity

N = Number of key-value pairs in the hash table.

	NO/FEW COLLISIONS	LOTS OF COLLISIONS
INSERT	$O(1)$	$O(N)$
FIND	$O(1)$	$O(N)$

Clustering Problem with High Collision Rates

Bad Hash Function: $\text{HASH}(\text{KEY}) = 5$

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		



$\text{HASH}(\text{KEY}) = 5$

Key Deletion



Finding key 32

Find value associated with key 32 = Fig

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 32 \% 10 = 2$$

Deletion

Delete key 13

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 13 \% 10 = 3$$

Deletion

Delete key 13

0	1	2	3	4	5	6	7	8	9
		2		34	32				
		Apple		Kiwi	Fig				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 13 \% 10 = 3$$

Finding key 32 After Deletion

Find value associated with key 32 = Not Found

0	1	2	3	4	5	6	7	8	9
		2		34	32				
		Apple		Kiwi	Fig				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 32 \% 10 = 2$$

Solution #1: Rehashing All Keys

Insert all keys into another empty hash table.

0	1	2	3	4	5	6	7	8	9
		2	32	34					
		Apple	Fig	Kiwi					

$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10$$

Solution #2: Lazy Deletion

Mark a hash entry as deleted without removing it.

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				
1	1	1	0	1	1	1	1	1	1

Finding key 32 After Deletion

Find value associated with key 32 = Fig

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				
1	1	1	0	1	1	1	1	1	1



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10 = 32 \% 10 = 2$$

Lazy Deletion using Exists Flag

```
struct HashEntry {  
    int key;  
    int value;  
    bool exists; // Indicates if the entry is active or deleted  
  
    // Updated constructor includes position  
    HashEntry(int k, int v, int pos) : key(k), value(v), exists(true) {}  
};
```

Vector to Static-Sized Array

```
class HashIndex {  
private:  
    static const size_t capacity = 100; // Hard-coded capacity  
    HashEntry hashTable[capacity];      // Static-sized array  
};
```


Position Tracking

Position is used for tracking the location of each entry within the hash table.

```
struct HashEntry {  
    int key;  
    int value;  
    int position; // Tracks the final position in the array  
    bool exists;  // Indicates if the entry is active  
  
    // Updated constructor includes position  
    HashEntry(int k, int v, int pos)  
        : key(k), value(v), position(pos), exists(true) {}  
};
```

Position Tracking

```
void insertOrUpdate(int key, int value) {  
    size_t originalIndex = hashFunction(key);  
    bool inserted = false;  
    do {  
        if (!hashTable[index].exists) {  
            hashTable[index] = HashEntry(key, value, true);  
            hashTable[index].position = index;  
            inserted = true;  
            break;  
        }  
        ...  
    }  
}
```


Limitations of Linear Probing

Clustering increases likelihood of collision; searches become inefficient

0	1	2	3	4	5	6	7	8	9
		2	12	22	32				
		Apple	Grape	Kiwi	Fig				



$$\text{HASH}(\text{KEY}) = \text{KEY} \% 10$$

Conclusion

- Database Configuration
- Indexing in C++
- Hash Table
- Hash Function
- Key Deletion