Lecture 2:
BuzzDB and C++

Recap

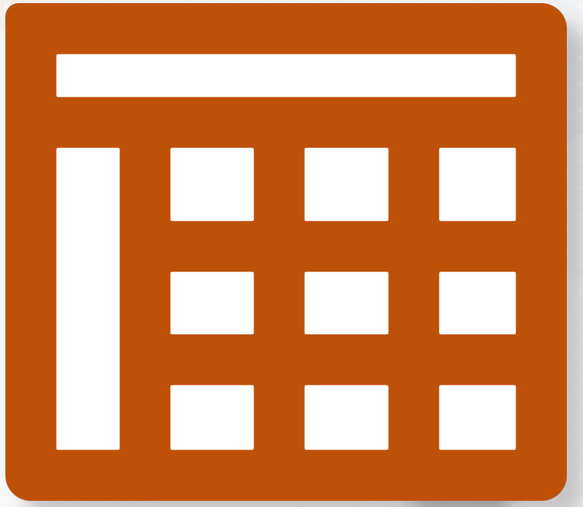
- Limitations of a Flat-File database system
- Benefits of a Relational database system

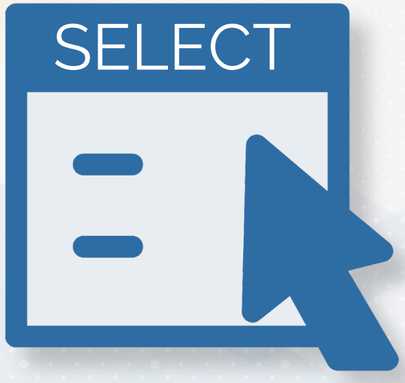
Lecture Overview

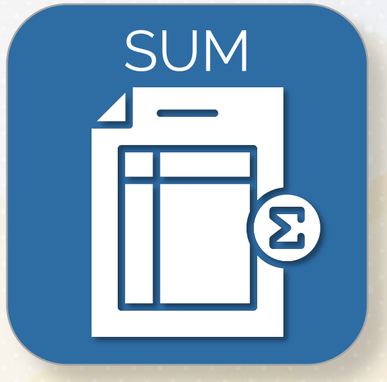
- Tour of relational operators
- BuzzDB
- Why C++?

Relational Operators

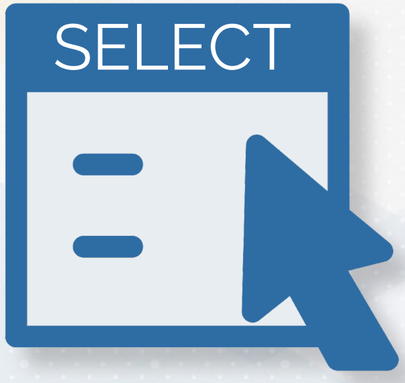
Relations

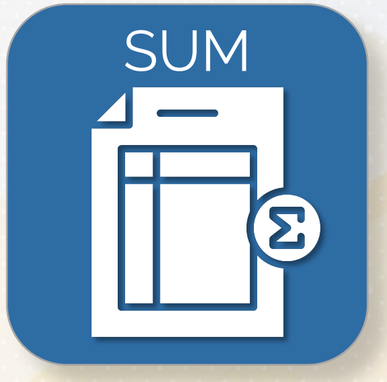




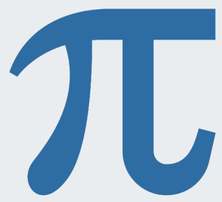


Relational Operators





SELECT (Projection Operator)



Select specific columns from a table

Example: Retrieve locations of all users.

```
SELECT Location
FROM Users;
```

WHERE (Selection Operator)

σ Filters rows based on specified conditions

Example: Find all interactions that are "Like" reactions.

```
SELECT *
FROM Interactions
WHERE ReactionType = 'Like';
```

GROUP BY (Grouping Operator)

γ

- Groups rows of same values
- Used with aggregate functions like SUM

Example: Count the number of reactions that each post received.

```
SELECT PostID,
       COUNT(*) AS ReactionCount
FROM Interactions
GROUP BY PostID;
```

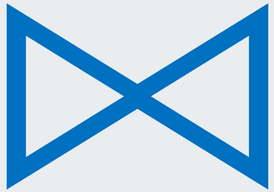
SUM (Aggregation Operator)

- Adds group values
- Defined by GROUP BY clause

Example: Total number of posts made by each user, grouping the results by UserID.

```
SELECT UserID,
    COUNT(PostID) AS TotalPosts
FROM Posts
GROUP BY UserID;
```

JOIN (Join Operator)



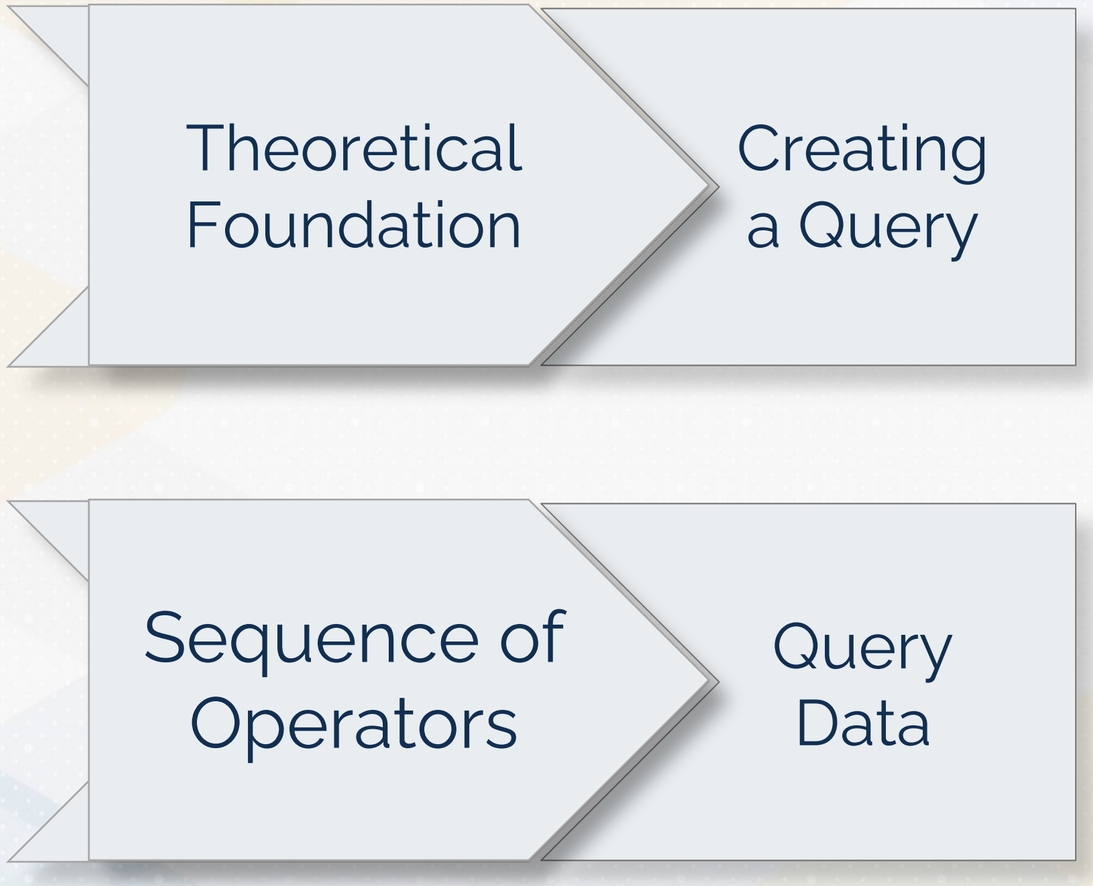
- Links rows from two different tables
- Combine information from both

Example: Total number of interactions each post receives.

```
SELECT Posts.PostID,
    COUNT(Interactions.ReactionType) AS TotalInteractions
FROM Posts
JOIN Interactions ON Posts.PostID = Interactions.PostID
GROUP BY Posts.PostID;
```

Relational Algebra

Relational Algebra



Relational Algebra

- Combine Operators
- Filter Interactions
- Combine Tables
- Group & Count Results
- Project Fields
- Sort Popular Posts

```
SELECT Interactions.PostID,
       COUNT(*) AS Likes,
       Users.UserID, Users.Username
FROM Interactions
JOIN Users ON Interactions.UserID =
Users.UserID
WHERE Interactions.ReactionType = 'Like'
GROUP BY Interactions.PostID,
         Users.UserID,
         Users.Username
ORDER BY Likes DESC;
```

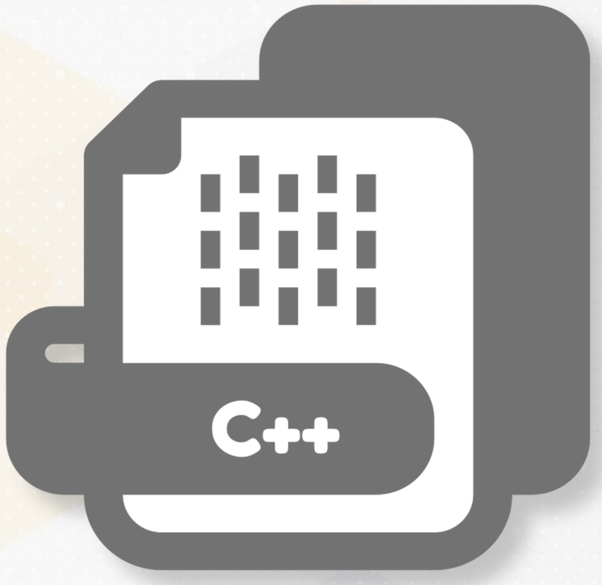
Relational Algebra

- Filters Interactions
- Combine Tables
- Group & Aggregate
- Project Output
- Order Posts

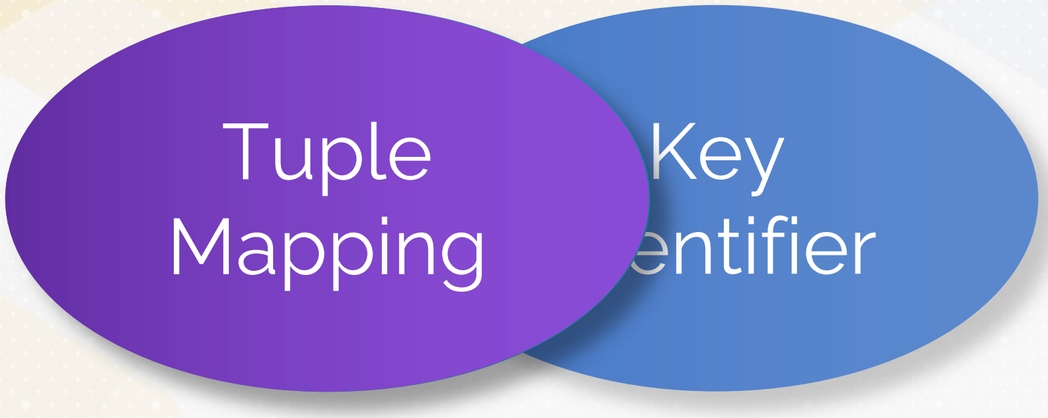
```
τ Likes DESC
 (π PostID, Likes, UserID, Username
  (γ PostID, UserID, Username;
COUNT(*)→Likes
   (σ ReactionType='Like'
    (Interactions) ⋈ Users)
   )
  )
```

BuzzDB

BuzzDB



Tuple Class




```
class Tuple {
public:
  int key;    // ID column
  int value;  // Data column
};
```

Classes in C++

Book Class

:apsulation

```
class Book {
  public:
    std::string title;
    std::string author;

    void displayInfo() {
      std::cout << "Title: " << title
                << "\nAuthor: " << author << std::endl;
    }
};
```

Classes in C++


```
Book myBook;                        // Create or instantiate a Book object
myBook.title = "1984";              // Set book title
myBook.author = "George Orwell"; // Set book author
myBook.displayInfo();               // Display book information
```

Classes in C++

Operational Functions

Organize Data

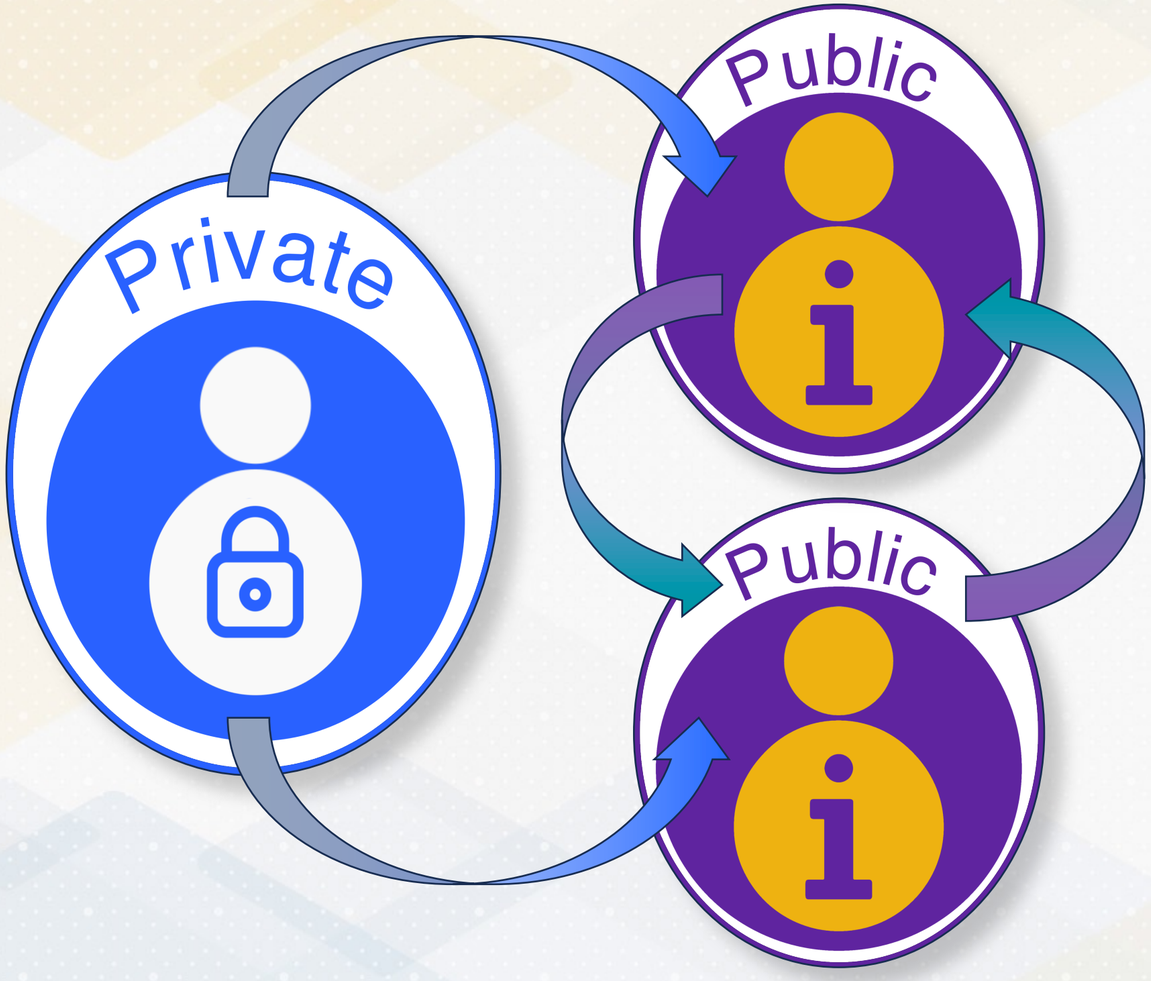
Modular + Reusable

Cleaner Code

Complex Data Creation

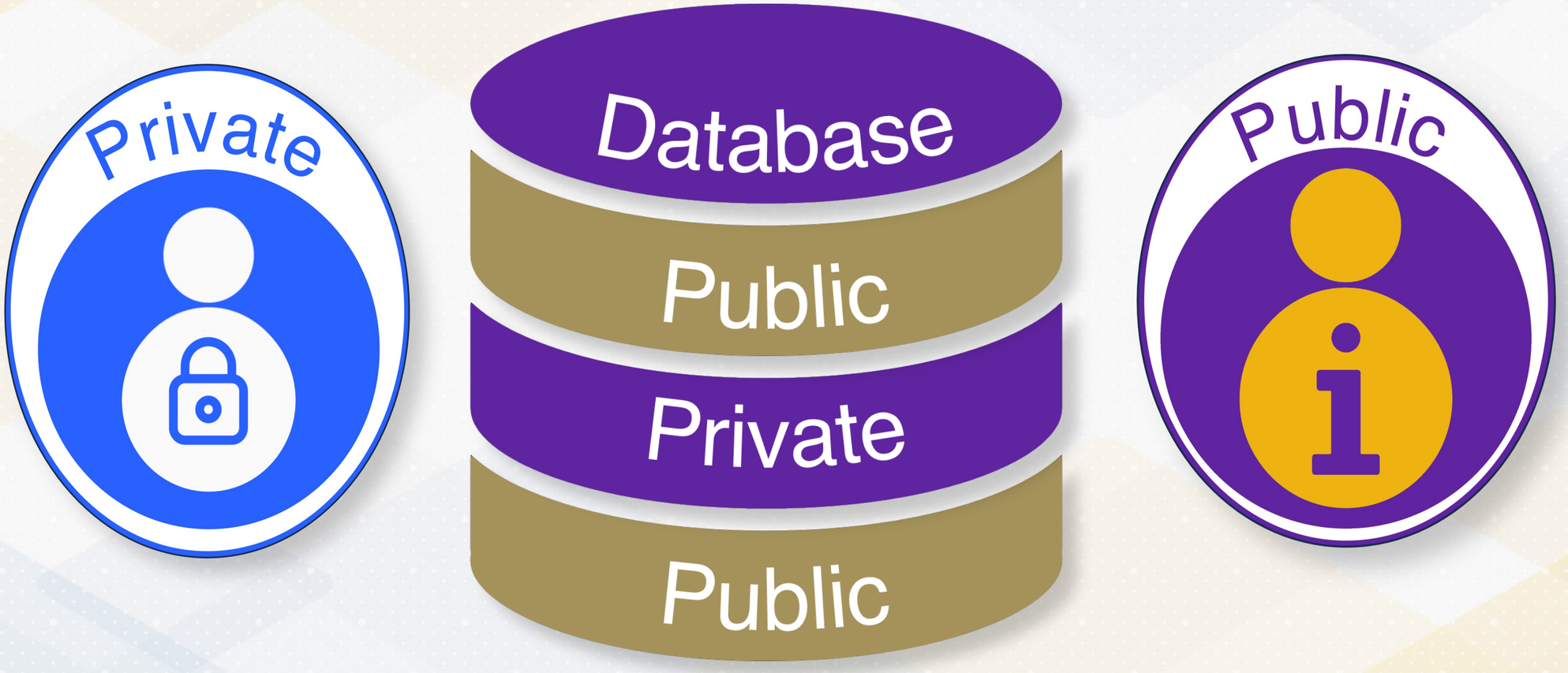
Create Book Type

Encapsulation



- Attribute Protection
- Public & Private
- Protected Access

Encapsulation



BuzzDB Class

Tuple Management

Vector Object Storage

Map Object Data Indexing

```
class BuzzDB {
public:
  // Stores all our Tuples
  std::vector<Tuple> table;


private:
  // Helps quickly find data by key
  std::map<int, std::vector<int>> index;
};
```

C++ Library

Vectors in C++

C++ Standard Library

std::vector in C++

Data Size Flexibility

```cpp
#include <vector>

// Declare a vector of integers
std::vector<int> myVector;
// Add an element to the end
myVector.push_back(10);
// Add another element
myVector.push_back(20);
// Access the first element (10)
std::cout << myVector[0];
```

Vectors in C++

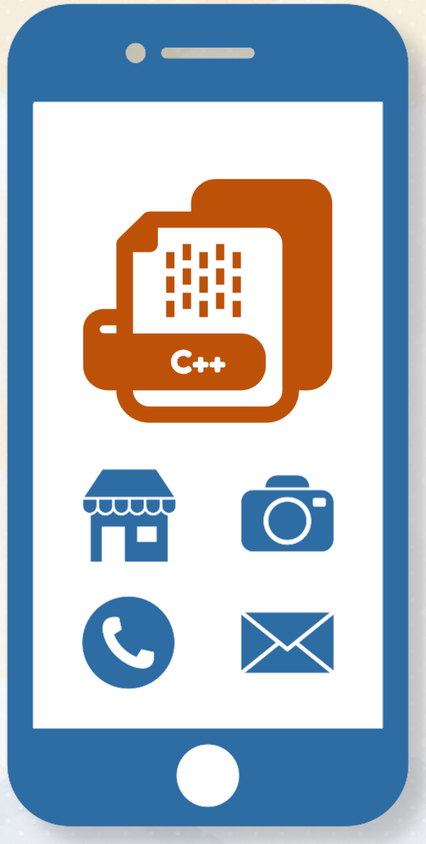
Vector Versatility

Vector Object Storage

Vector Storage & Retrieval

```
// Declare a vector for Tuple objects
std::vector<Tuple> tuples;

// Adding Tuple objects to our vector
tuples.push_back(Tuple(1, 100));
tuples.push_back(Tuple(2, 200));
// Accessing and displaying elements of the vector
for (const auto &tuple : tuples) {
  std::cout << "Key: " << tuple.key << ", Value: " << tuple.value << std::endl;
}
```

Maps in C++



Placeholder Function

Links Keys & Values

```cpp
#include <map>
// Declare a map with string keys and int values
std::map<std::string, int> myMap;
myMap["apple"] = 5;   // Assign 5 to key "apple"
myMap["banana"] = 10; // Assign 10 to key "banana"
// Access the value associated with "apple" (5)
std::cout << myMap["apple"];
```

Maps in C++

Keys Associated in Order

Int = Key Type

```
// Declaring a map with an int key and vector<int> as value
std::map<int, std::vector<int>> index;


// Adding data to the map
index[1].push_back(100);
index[1].push_back(200); // Multiple values under the same key
index[2].push_back(50);
```

Maps in C++

```
for (const auto &pair : index) {
    std::cout << "Key: " << pair.first << ", Values: ";
    for (int value : pair.second) {
      std::cout << value << ", ";
    }
    std::cout << std::endl;
}

// PROGRAM OUTPUT
Key: 1, Values: 100, 200
Key: 2, Values: 50
```

BuzzDB Operators

Insertion Operator in C++

UserName,	Location
Timothée Chalamet,	Paris

```
void BuzzDB::insert(int key, int value) {
    Tuple newTuple = {key, value};
    table.push_back(newTuple);   // Add to main table vector
    index[key].push_back(value); // Also, update the index map
 }
```

Key-Based Tuple Retrieval

Populating the Database

BuzzDB Insert Method Creation

```
int main() {
    BuzzDB db;
    // Populating the database
    db.insert(1, 100); db.insert(1, 200);
    db.insert(2, 50);
    db.insert(3, 200); db.insert(3, 200); db.insert(3, 100);
    db.insert(4, 500);
    // Executing aggregation query
    db.selectGroupBySum();
    return 0;
 }
```

Aggregation Query

selectGroupBySum method

Tally Summarization

Iterate Over Keys

Iterate Over Values

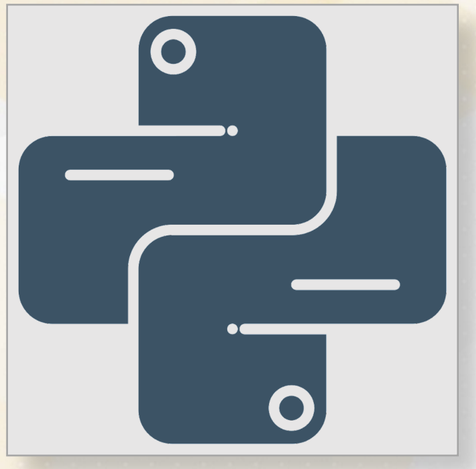
```
void BuzzDB::selectGroupBySum() {
  // Iterate over each key
  for (auto const &pair : index) {
    int sum = 0;
    // Sum values for this key
    for (auto const &value : pair.second) {
      sum += value;
    }
    std::cout << "key: " << pair.first << ", sum: " << sum
<< '\n';
  }
}
```

Aggregation Query


```
int main() {
    ...
    // Executing aggregation query
    db.selectGroupBySum();
    return 0;
}
// PROGRAM OUTPUT
key: 1, sum: 300
key: 2, sum: 50
key: 3, sum: 500
key: 4, sum: 500
```

Why C++?

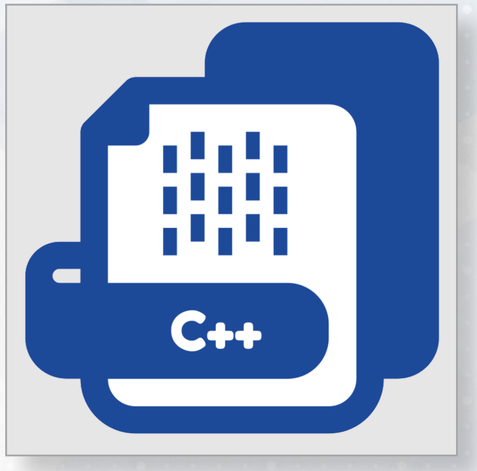
C++ vs Python



Superior Performance

Executable Code

Fast & Efficient



Performance: C++

10^8 integers	ff = Time

```cpp
#include <chrono>
#include <iostream>

int main() {
  auto start = std::chrono::high_resolution_clock::now();
  long sum = 0;
  for (int i = 0; i < 100000000; ++i) {
    sum += i;
  }
  auto end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> diff = end - start;
  std::cout << "Time taken: " << diff.count() << " seconds\n";
}
```

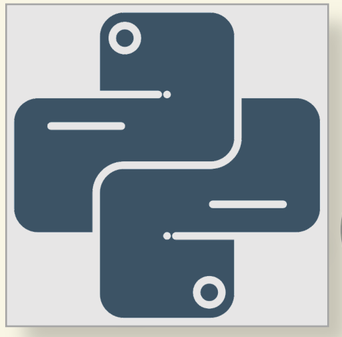
Performance: Python

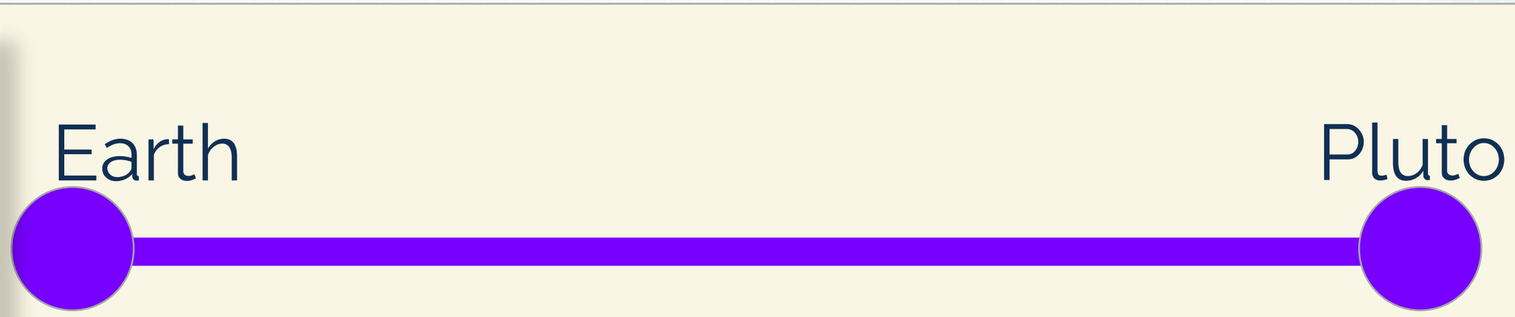
5.24 Seconds

terpreted
xecution

```
import time

start = time.time()

sum = 0
for i in range(100000000):
    sum += i

end = time.time()
print(f"Time taken: {end - start} seconds")
```

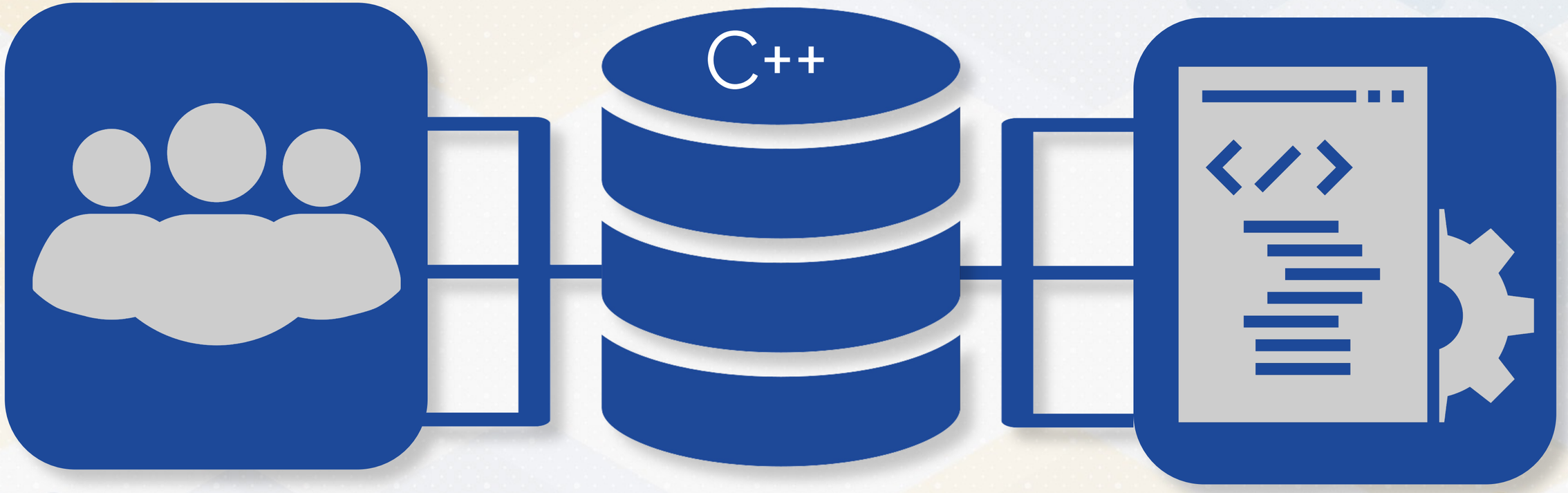
Performance Comparison





83220 ÷ 2 = 41610

C++ vs Python



C++ Memory Management

std::unique_ptr

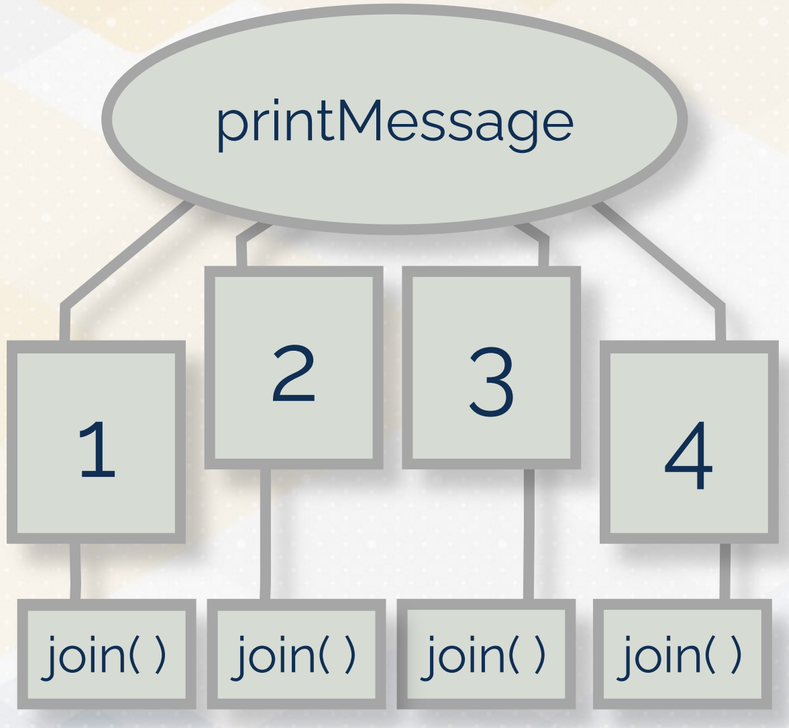
allocateArray

Memory Leaks

Smart Pointers

```cpp
#include <memory>
std::unique_ptr<int[]> allocateArray(int size) {
  return std::make_unique<int[]>(size);
}
int main() {
  auto myArray = allocateArray(1000000); // Allocate memory
  for (int i = 0; i < 1000000; ++i) {
    myArray[i] = i;
  }
  std::cout << "Array[500000] = " << myArray[500000] << std::endl;
  // No need to manually delete,
  // memory is automatically freed when out of scope
}
```

C++ Multi Threading



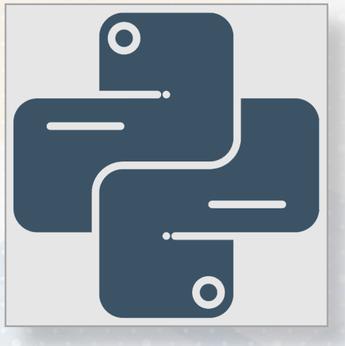

```cpp
#include <thread>

// Function that each thread will execute
void printMessage(int threadId) {
  std::cout << "Hello from Thread " << threadId << std::endl;
}
int main() {
  const int numThreads = 4;          // Number of threads to create
  std::vector<std::thread> threads; // Vector to hold all threads
  // Create and start threads
  for (int i = 0; i < numThreads; ++i) {
threads.emplace_back(printMessage, i + 1); }
  // Wait for all threads to finish
  for (auto &t : threads) { t.join();}
}
```

C++ vs Python: Typing

Code Stability

Performance

```cpp
#include <iostream>
#include <string>
int main() {
  int myInt = 10;  // Type must be specified
  myInt = "Hello"; // Compile-time error
}
```



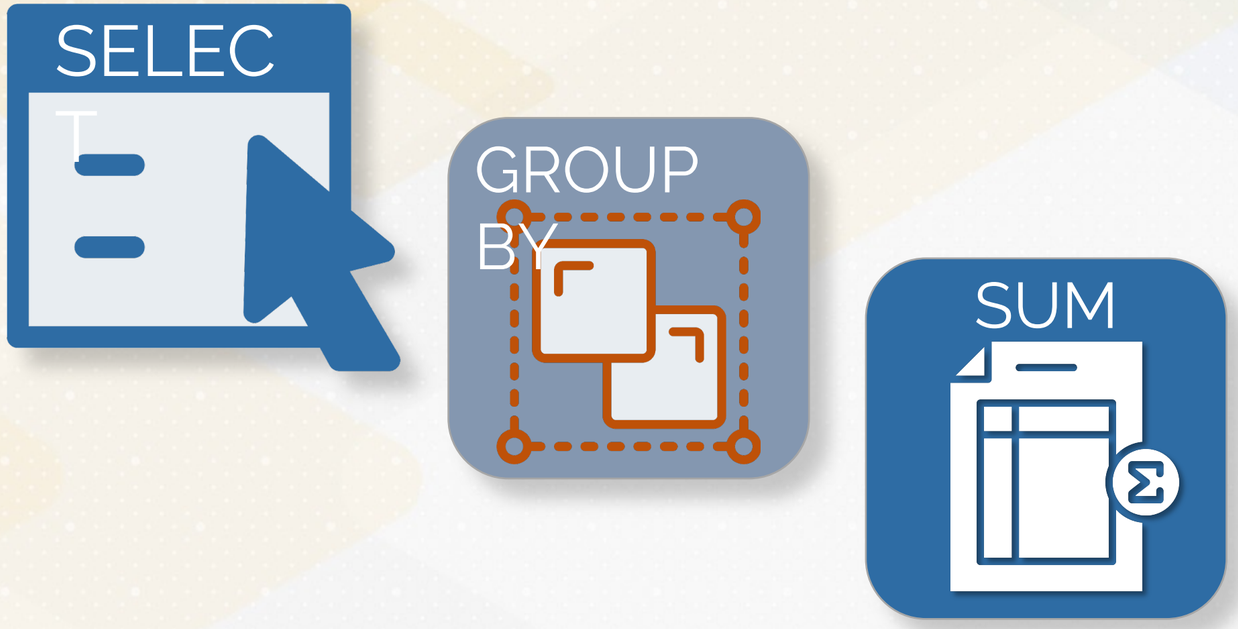
Flexible Type

Errors & Inefficiencies

```python
my_var = 10       # Initially an int
my_var = "Hello"  # Can be reassigned to a string without error
```

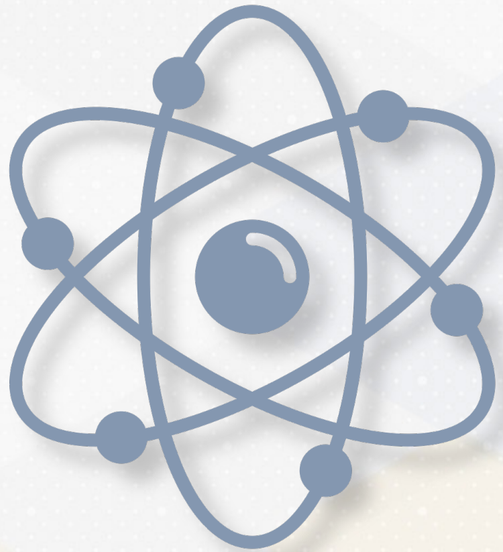
Transactions

Queries vs Transactions



- ✓ Multiple Database Changes

Transaction: Atomicity Property



Transaction: Atomicity Property

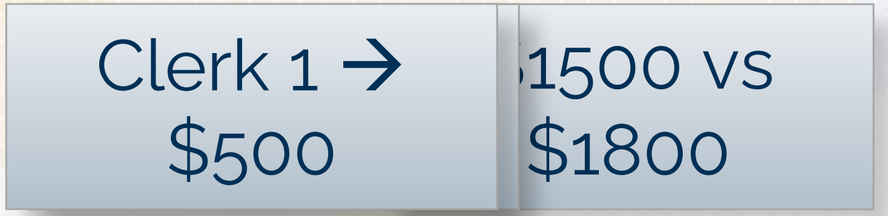
Transaction: Consistency Property

$100 from Account 1 to Account 2

Update Failure → No Partial Updates

```
BEGIN TRANSACTION;

-- Withdraw $100 from account 1
UPDATE ACCOUNT SET Balance = Balance - 100 WHERE AccountID = 1;

-- Deposit $100 into account 2
UPDATE ACCOUNT SET Balance = Balance + 100 WHERE AccountID = 2;


COMMIT;
```

Transaction: Isolation Property

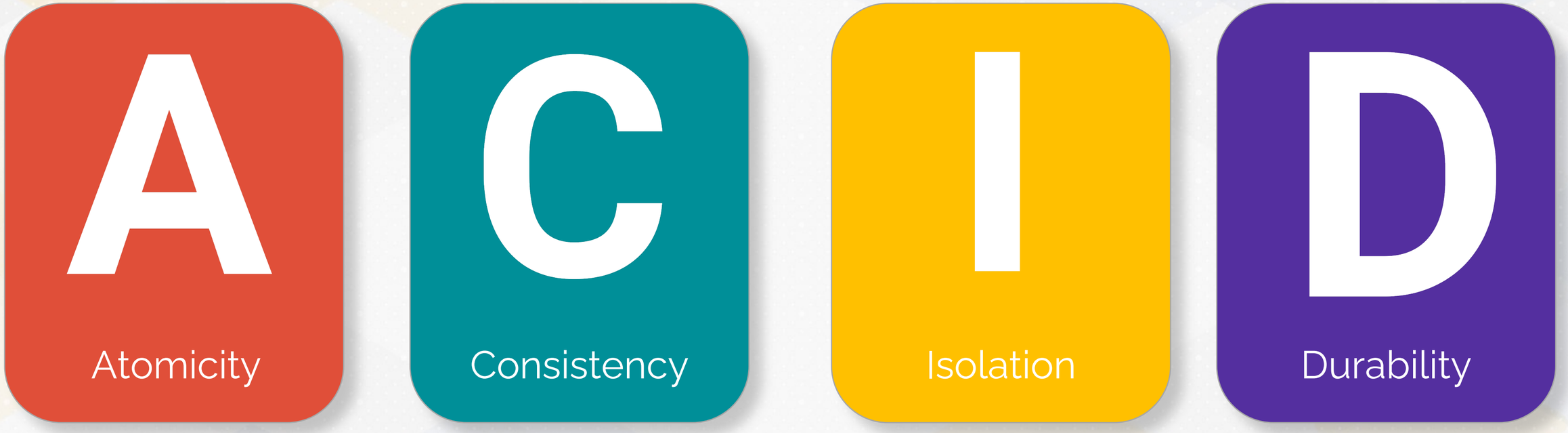



```
-- Clerk 1: Deposits $500 into account 1
BEGIN TRANSACTION;
SELECT Balance FROM ACCOUNT WHERE AccountID = 1; -- Suppose it returns $1000
UPDATE ACCOUNT SET Balance = 1000 + 500 WHERE AccountID = 1;
COMMIT;
-- Clerk 2: Deposits $300 into account 1 almost at the same time
BEGIN TRANSACTION;
SELECT Balance FROM ACCOUNT WHERE AccountID = 1; -- Suppose it still returns $1000
UPDATE ACCOUNT SET Balance = 1000 + 300 WHERE AccountID = 1;
COMMIT;
```

Transaction: Isolation Property

```
BEGIN TRANSACTION;
UPDATE ACCOUNT SET Balance = Balance - 500 WHERE AccountID = 1; -- Customer withdraws $500
COMMIT;
```

ACID Properties



History of “ACID”

Andreas Reuter (1983)

Reliable Management

Multi-User Capability

Photo: Gülay Keskin/Heidelberg Institute for Theoretical Studies (HITS)

Conclusion

- Tour of relational operators
- BuzzDB and C++
- Transactions and ACID properties