Lecture 15:
Trie

Logistics

- Programming assignment 3 (B+Tree) due on **Nov 2**
- Two-page project updates due on **Oct 29** (extra credit)

Recap

- B+Tree Structure
- Range Query Processing
- Node Split

Lecture Overview

- Trie
- Binary Patricia Trie
- Modular Query Execution

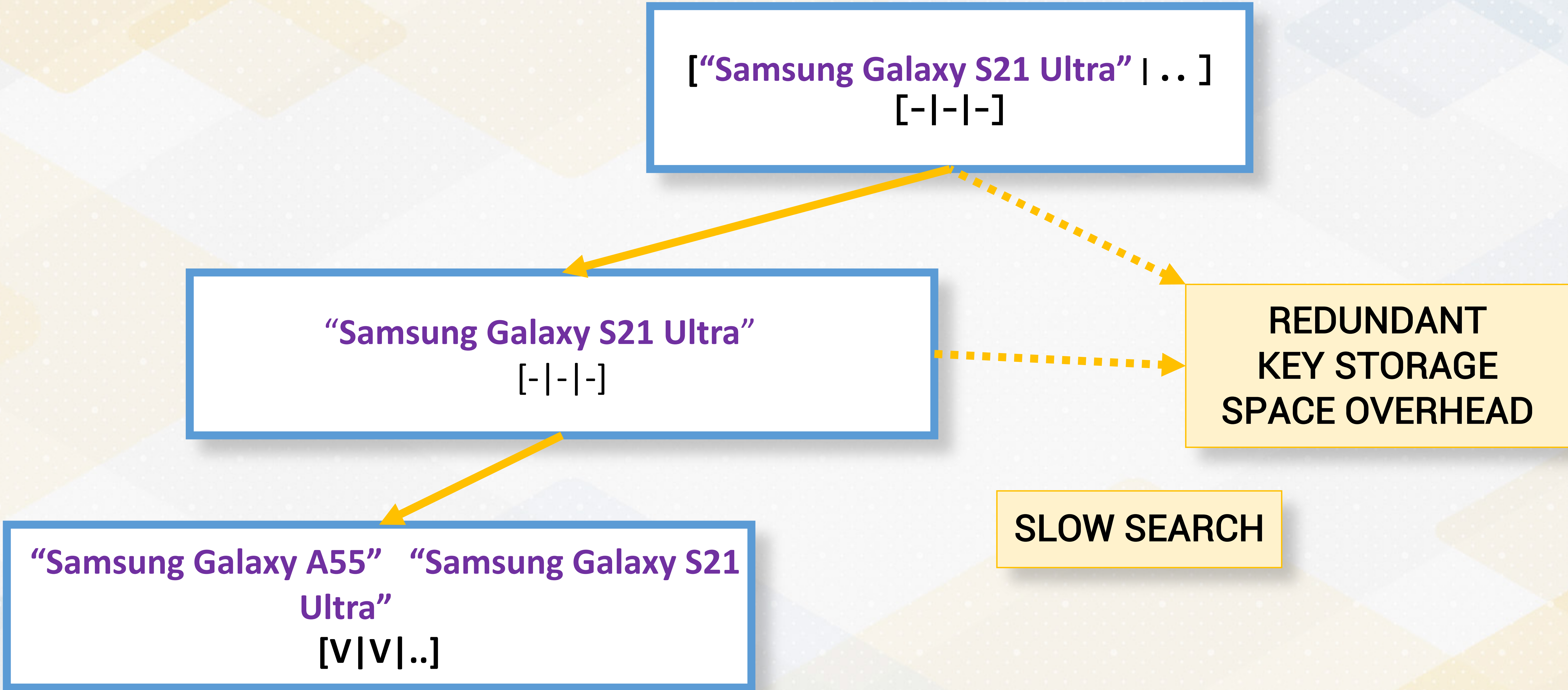
Trie

Product Catalog Search

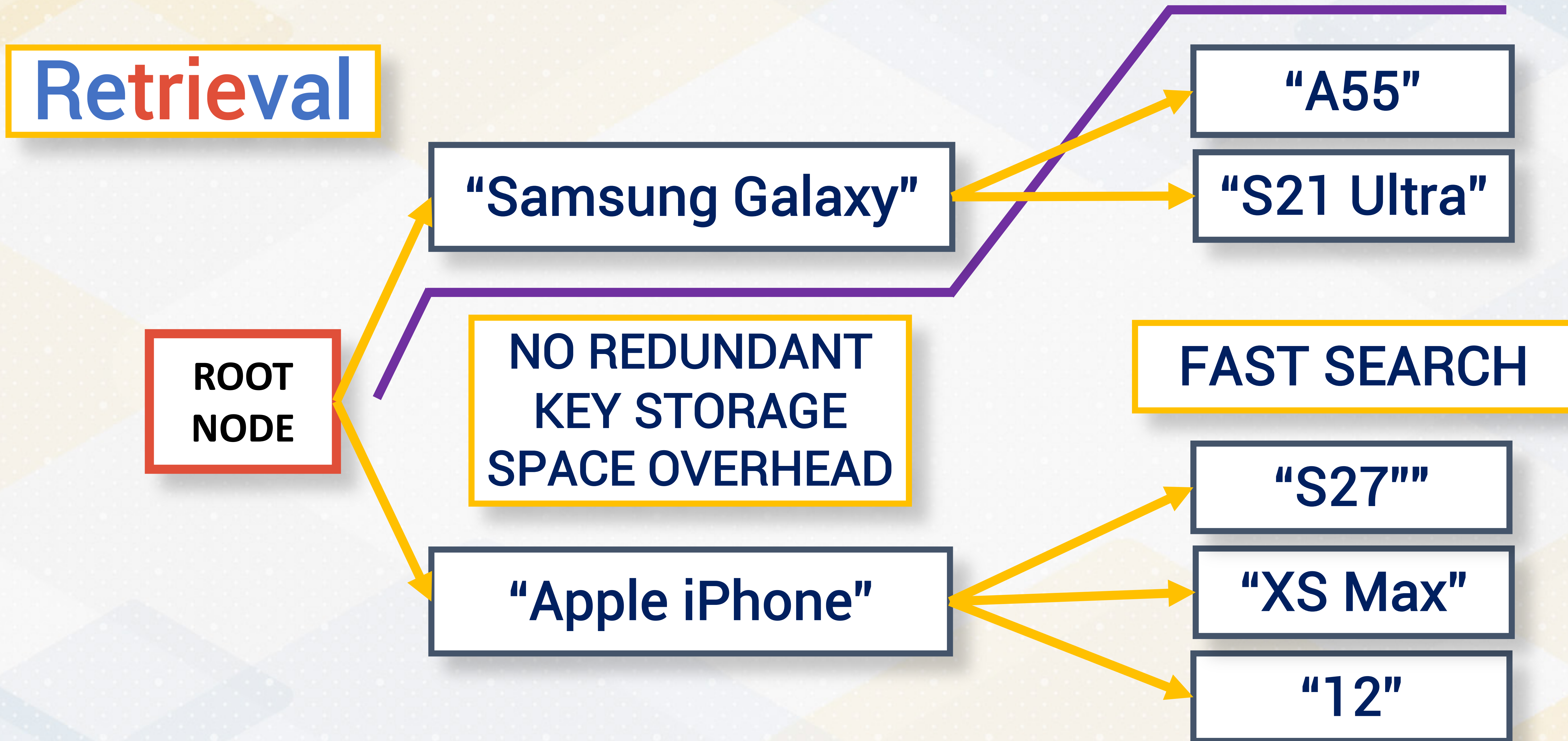
Product Name	Relevant or not
Samsung Galaxy S21 Ultra	Yes
Apple iPhone XS Max	No
Samsung Galaxy A55	Yes
Apple iPhone 12	No
Samsung Galaxy S27	Yes

```
// Find all Samsung Galaxy phones in the catalog
std::vector<std::string> galaxyPhones =
productCatalog.startsWith("Samsung Galaxy");
```

Limitations of B+Tree



Trie



Patricia Trie

Trie

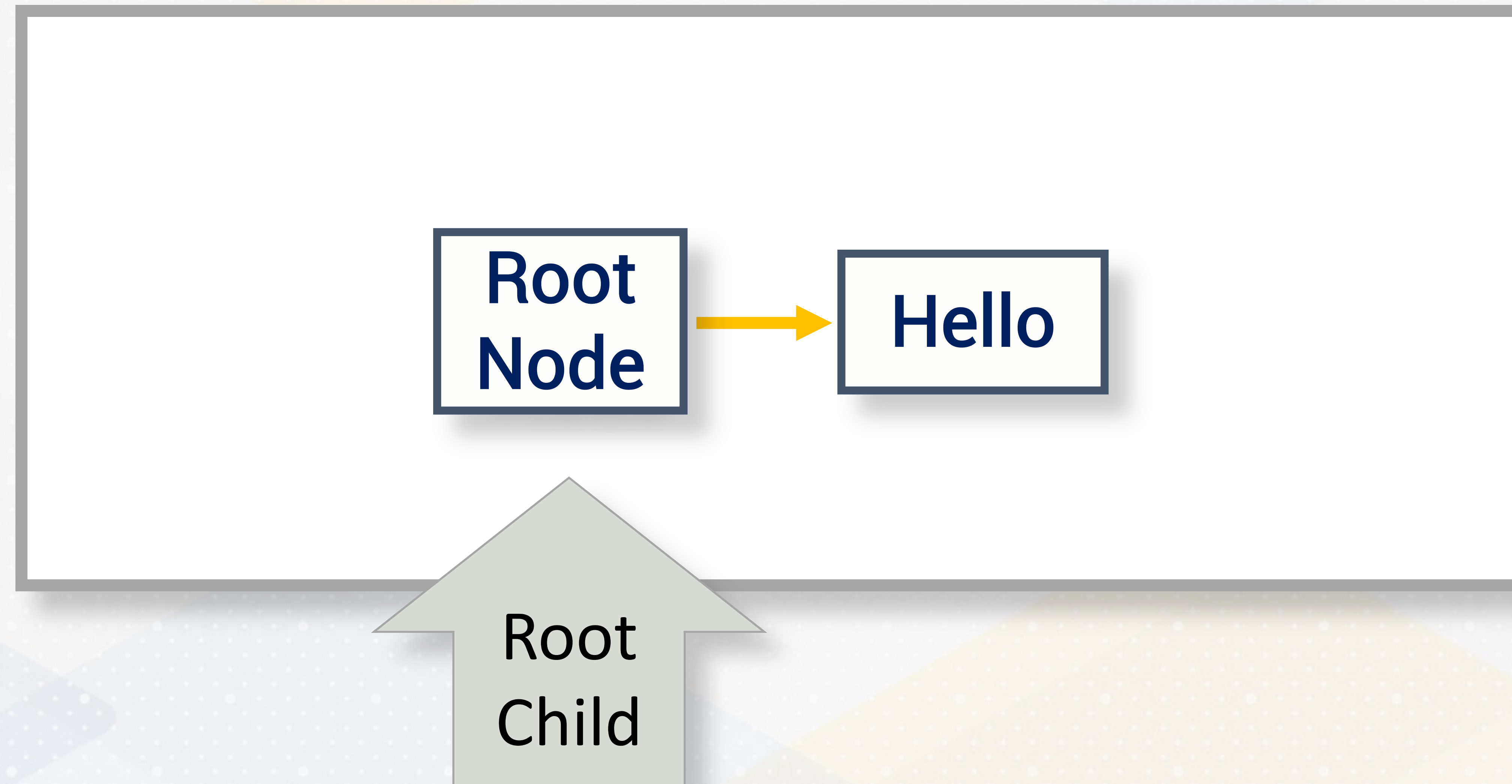
Merge Single Character Nodes

Whole Strings

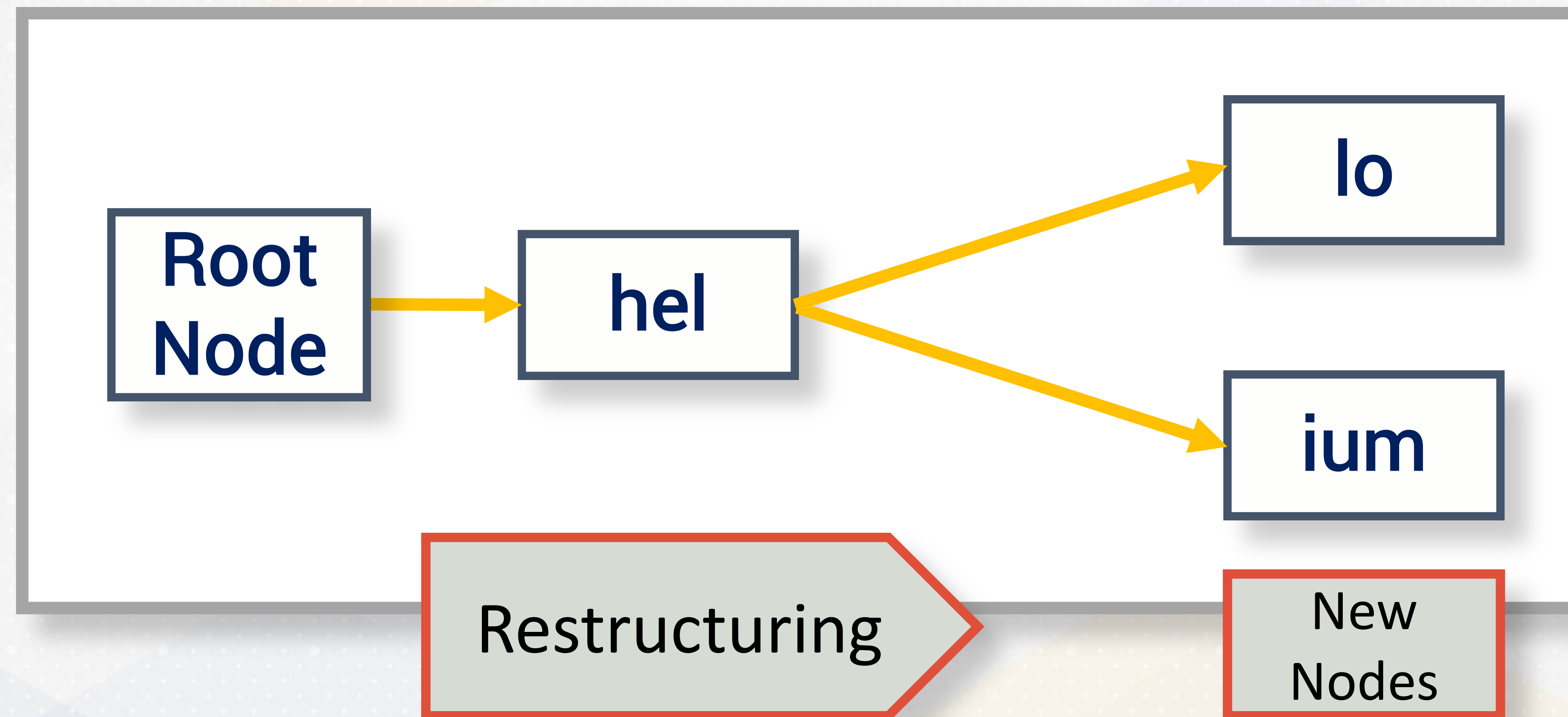
Patricia Trie

```
class PatriciaNode {
public:
    bool isEndOfWord;
    std::map<std::string, PatriciaNode*> children;

    PatriciaNode() : isEndOfWord(false) {}
};
```

Insertion Example



Insertion Example

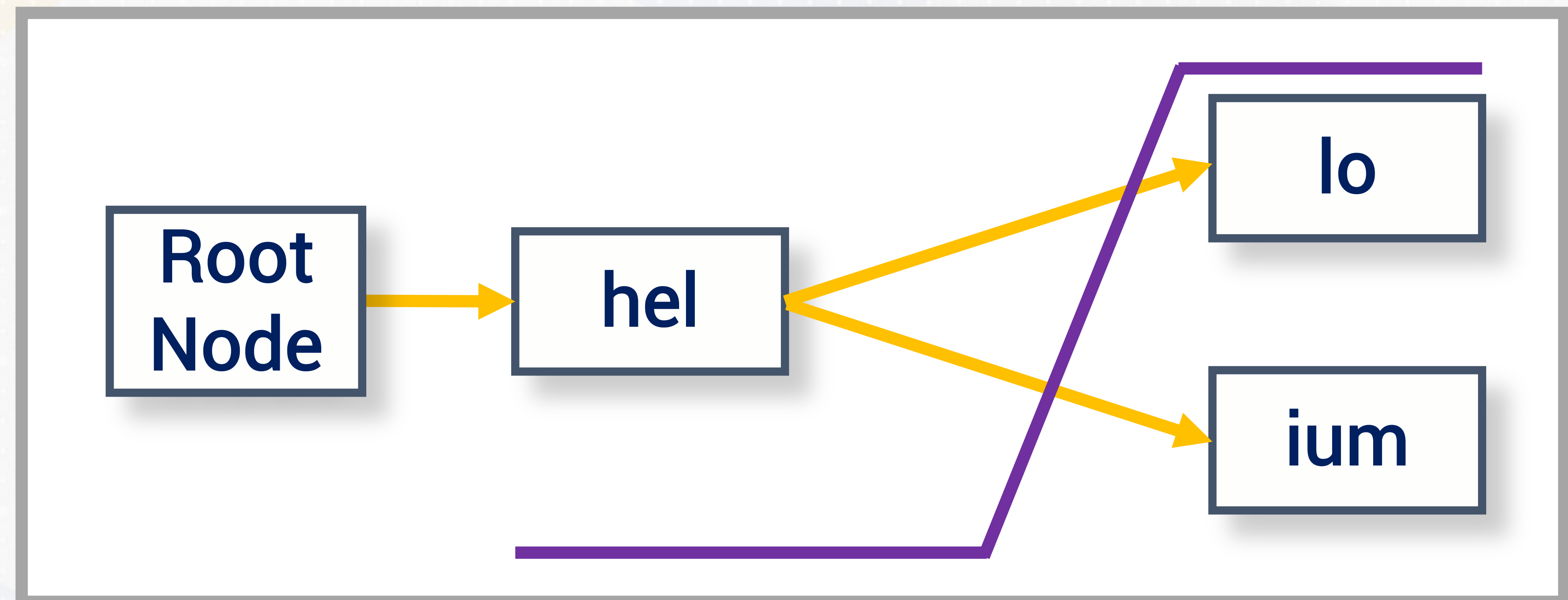


Retrieval

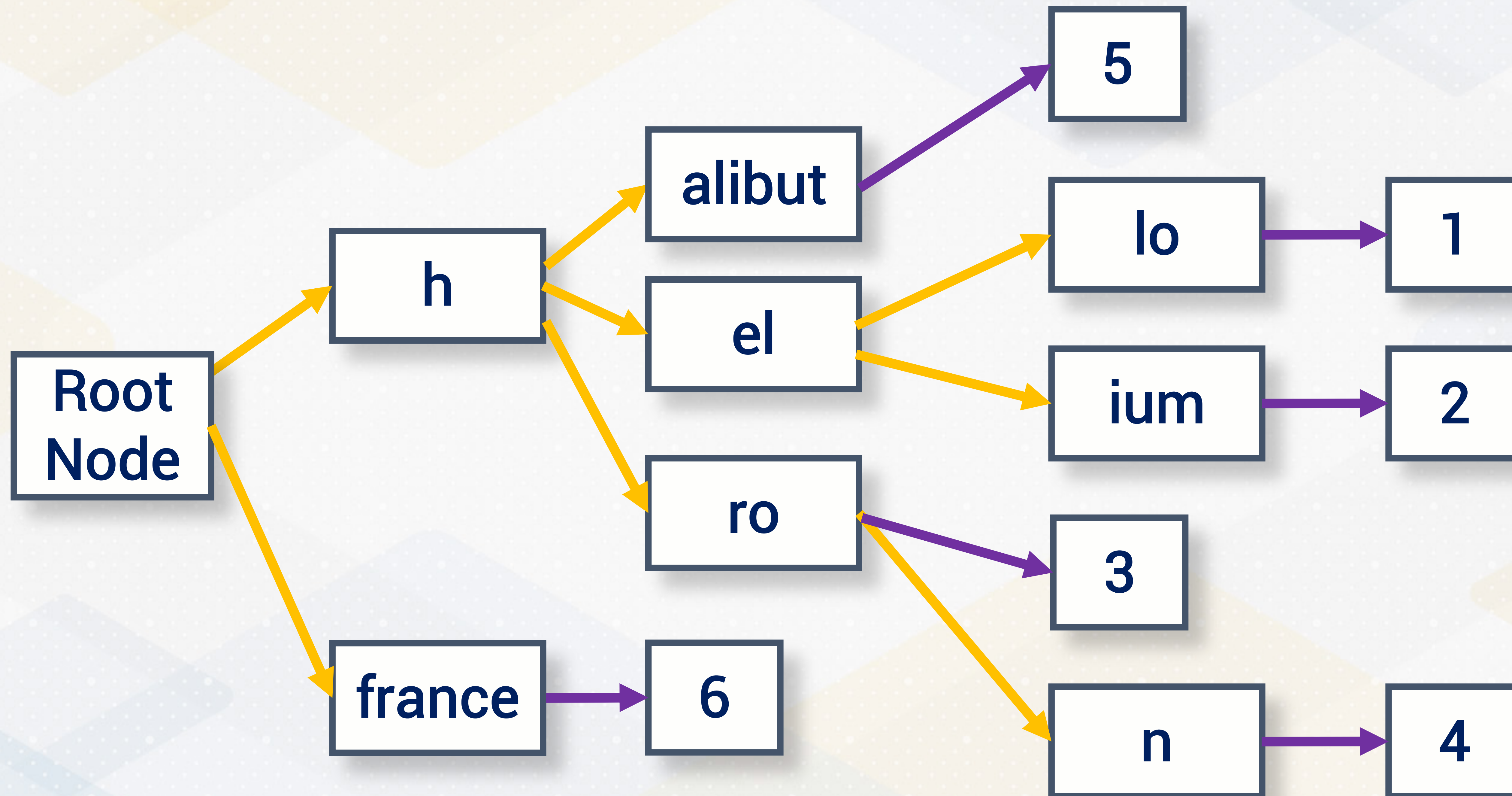
Navigate Root Node to Relevant Leaf

Each Character Narrows Search Path

Node Visits Narrow Search to Matching Key



Larger Trie with Values



Insertion

Insertion Check

isEndOfWord = True

Stops Recursion

```
void insertHelper(PatriciaNode* node, const std::string& word,
size_t index, int value) {
    // Base case: If the whole word has been processed
    if (index == word.length()) {
        node->isEndOfWord = true;
        return;
    }

    // Prepare the substring from the current index to the end
    std::string remaining = word.substr(index);
```

Insertion: Identifying Common Prefixes

Trie

Identifies Common Prefixes

Calculates Common Prefix Lengths

```
 for (auto& child : node->children) {
        const std::string& key = child.first;
        PatriciaNode* childNode = child.second;

        // Calculate common prefix length
        size_t commonLength = 0;
        while (commonLength < key.length() && commonLength <
remaining.length() &&
               key[commonLength] == remaining[commonLength]) {
            commonLength++;
        }
```

Node Splitting

```
if (commonLength > 0) { // If a common prefix is found
 if (commonLength < key.length()) {
    // Node needs to be split
    std::string newKey = key.substr(0, commonLength);
    std::string oldKey = key.substr(commonLength);
    PatriciaNode* newChildNode = new PatriciaNode();
    newChildNode->children[oldKey] = childNode; // Move the old node
under the new node
    node->children[newKey] = newChildNode; // Insert the new node
    node->children.erase(key); // Remove the old node
    // Continue insertion in the new node structure
    insertHelper(newChildNode, word, index + commonLength, value);
}
```

Insertion: Handling No Common Prefix

```
 // If no child matches the remaining part of the word
 node->children[remaining] = new PatriciaNode();
 // Recursively insert the remainder
insertHelper(node->children[remaining], word, word.length(), value);
```

Complexity

- L: Length of the key
- N: Number of keys

	Patricia Trie	B+Tree
Insertion	O(L)	O(log N)
Search	O(L)	O(log N)

Use Case

	Patricia Trie	B+Tree
Target Use Case	Fast prefix and key-value lookups	Large datasets, efficient range queries
Memory Efficiency	Less efficient for sparse keys	More efficient, stores many keys per node
Range Scans	Supported but not efficient	Highly efficient

Binary Patricia Trie

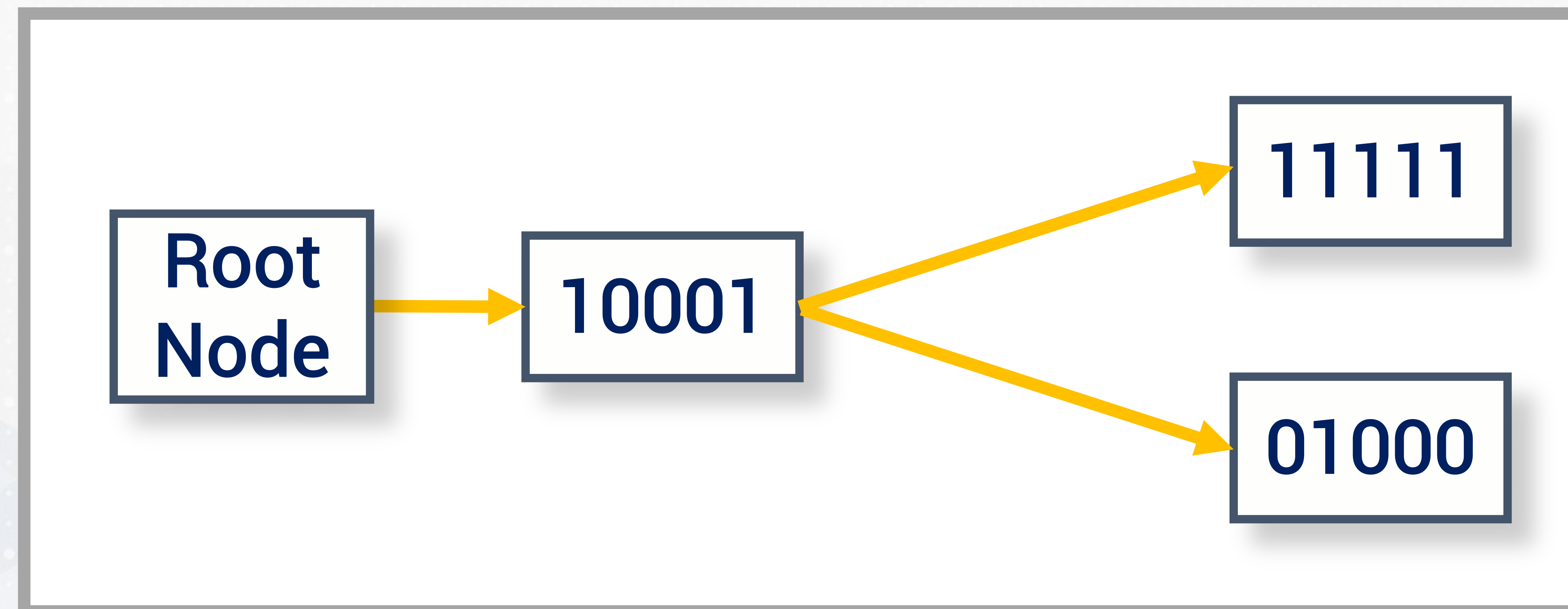
Binary Patricia Trie

Binary Patricia Trie

Processes Whole Character Strings

Designed for Bit-Level Binary Data

Not for Regular Character Strings



Radix of the Trie

Radix — the number of unique symbols or characters each node can handle.

	Radix	Characters
Binary Radix Trie	2	0 or 1
Ascii Radix Trie	256	'a', ' ', '-' etc.

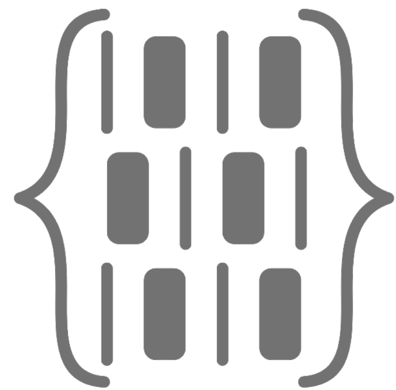
std::bitset

C++, bitset::to_string() function: converts text to binary string for insertion into Binary Patricia Trie

“hi”

01101000 01101001 00001010

std::bitset



convertToBinary

Converts Text String

Creates 8-bit Binary Representation

```
#include <bitset>
// Convert a string to its binary ASCII representation
std::string convertToBinary(const std::string& text) {
    std::string binaryString;
    for (char c : text) {
        binaryString += std::bitset<8>(c).to_string();
// Each char to an 8-bit binary
    }
    return binaryString;
}
```

Modular Query Execution

Limitations of Hard-Coded Query

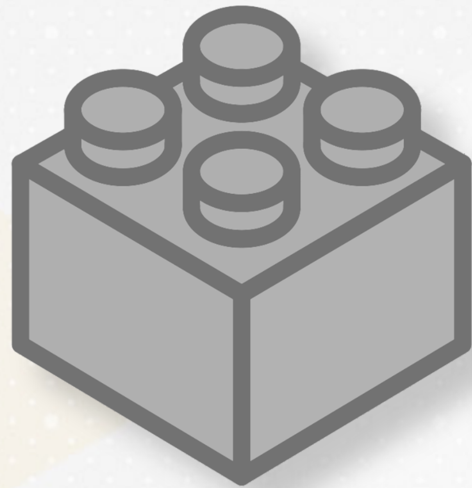
Hard to Change Query

Tight Coupling

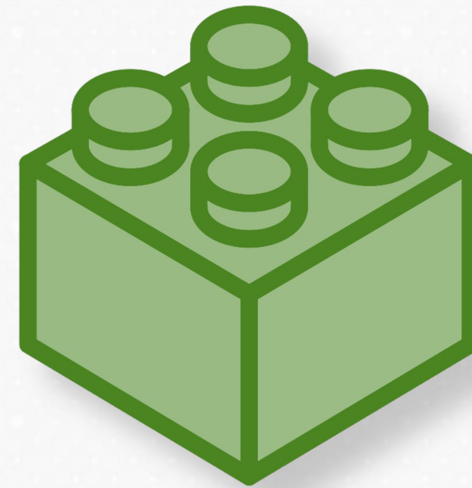
```
// perform a SELECT ... GROUP BY ... SUM query
void selectGroupBySum(int lowerBound, int upperBound) {
        hash_index.print();


        auto results = hash_index.rangeQuery(lowerBound, upperBound);
        std::cout << "Results: " << results.size() << "\n";
}
```

Modular Query Execution

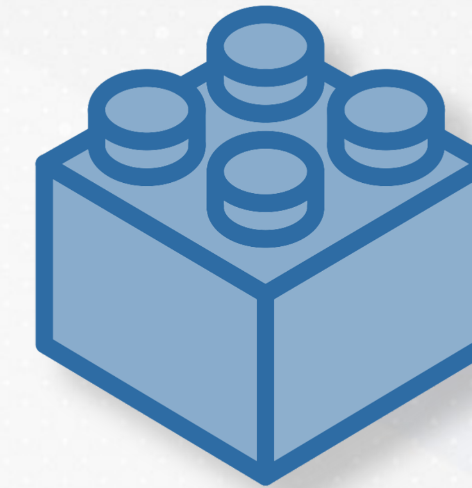
Scan Operator



Select Operator



Group By Operator



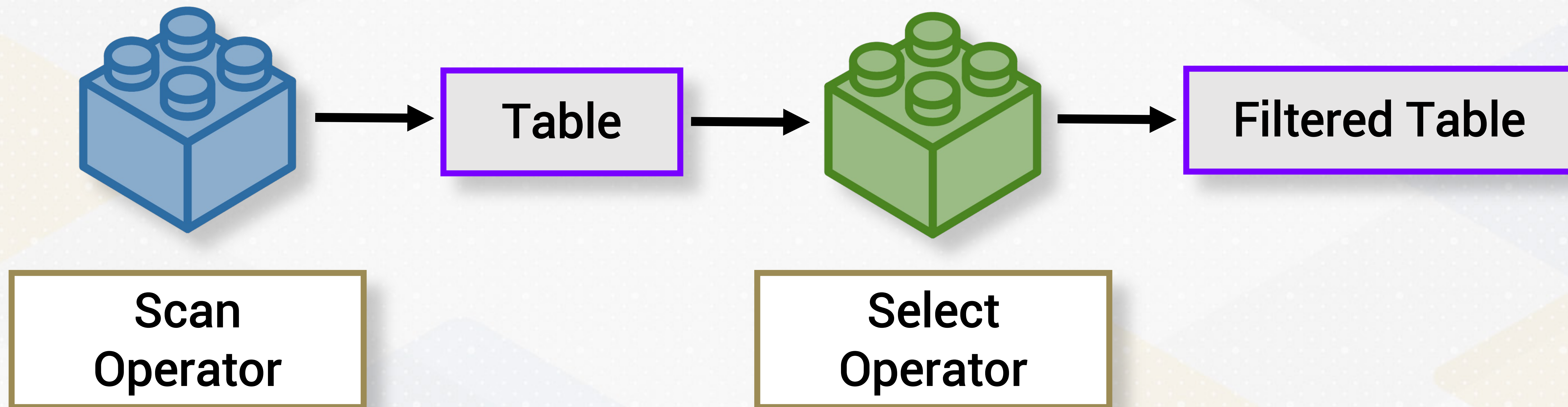
Flexibility

Flexible Query Configuration

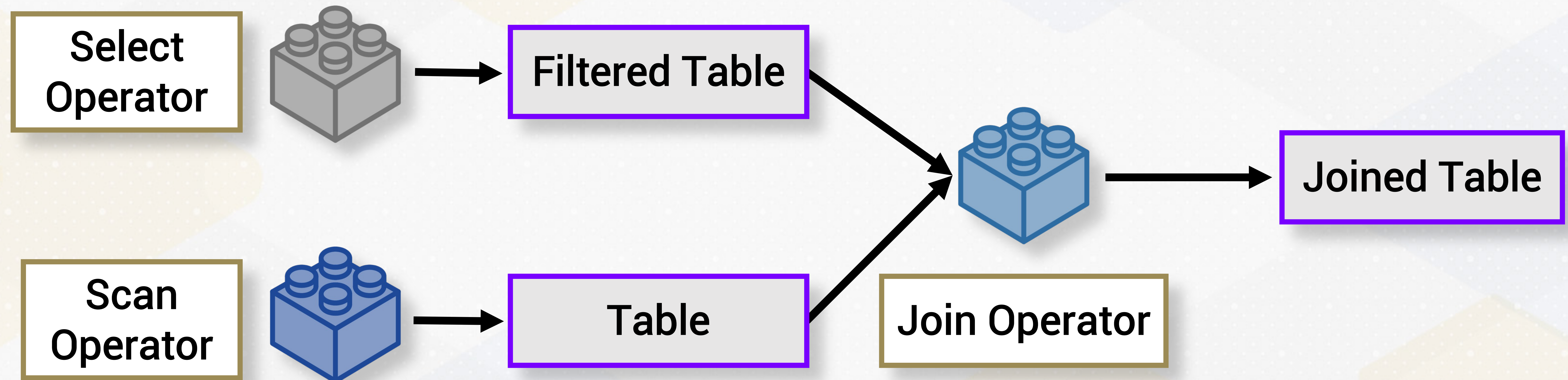
Isolation

Operator Changes Isolated

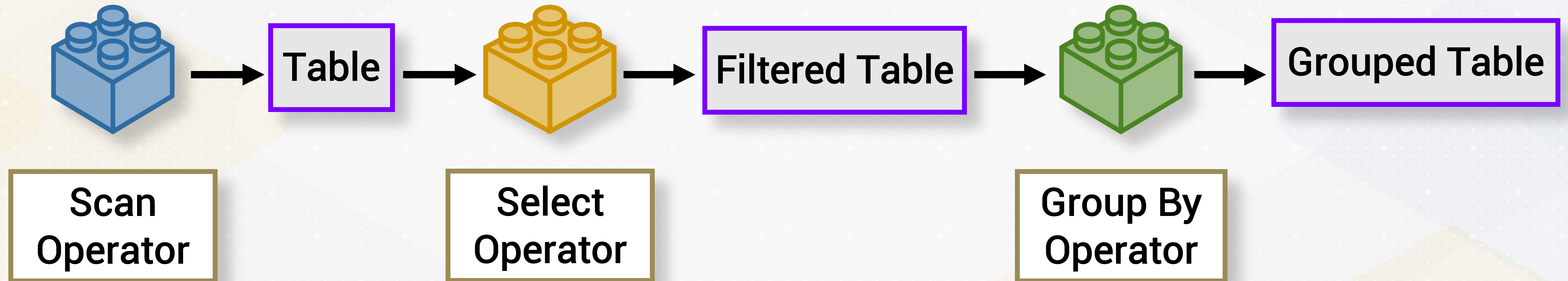
Operators are like Lego Blocks



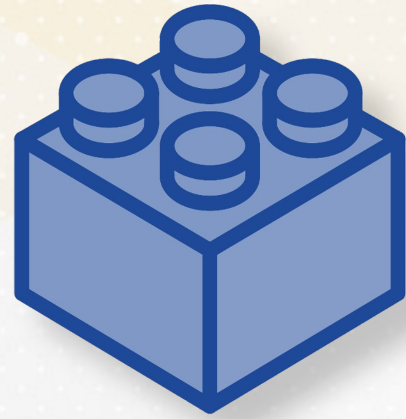
Composability of Operators



Composability of Operators

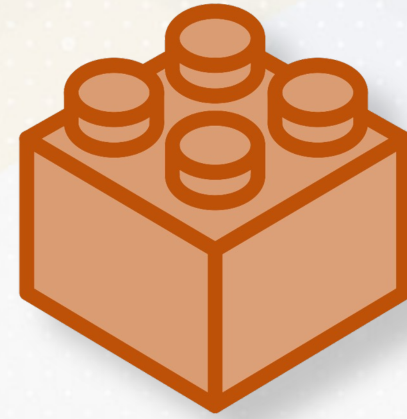


Benefits of Operators



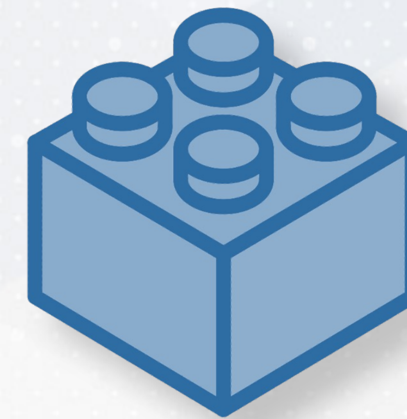
Flexibility

Operators process queries without altering the individual operator



Modularity

Each operator encapsulates a specific data manipulation functionality



Reusability

Reduce redundancy and minimize the chance of bugs

GT

Conclusion

- On-disk B+Tree
- Trie
- Binary Patricia Trie
- Modular Query Execution