

Lecture 15:

Trie



Logistics

- Assignment 3 has been released
- Clarifications related to extra-credit project

Recap

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

Lecture Overview

- Trie
- Binary Patricia Trie
- Inverted Index
- Web-Scale Search

Trie

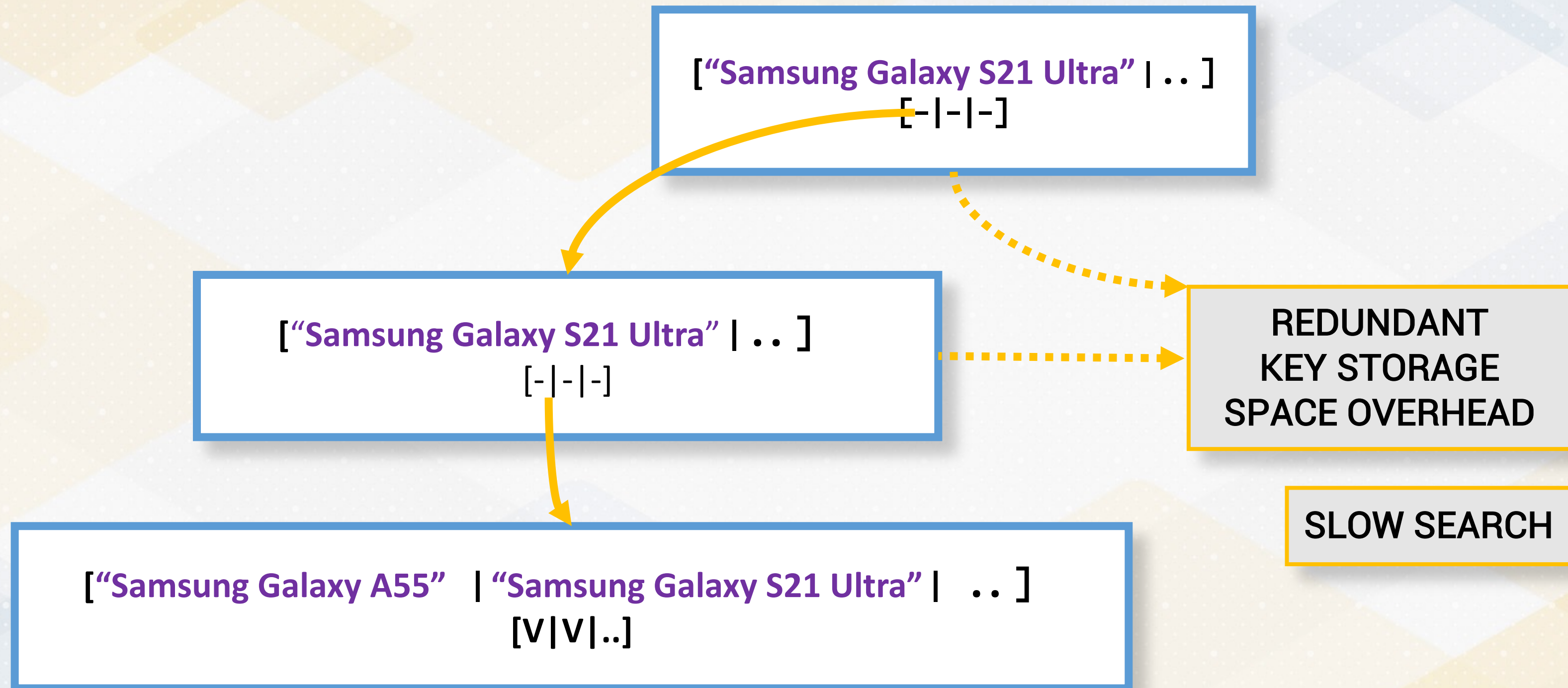


Product Catalog Search

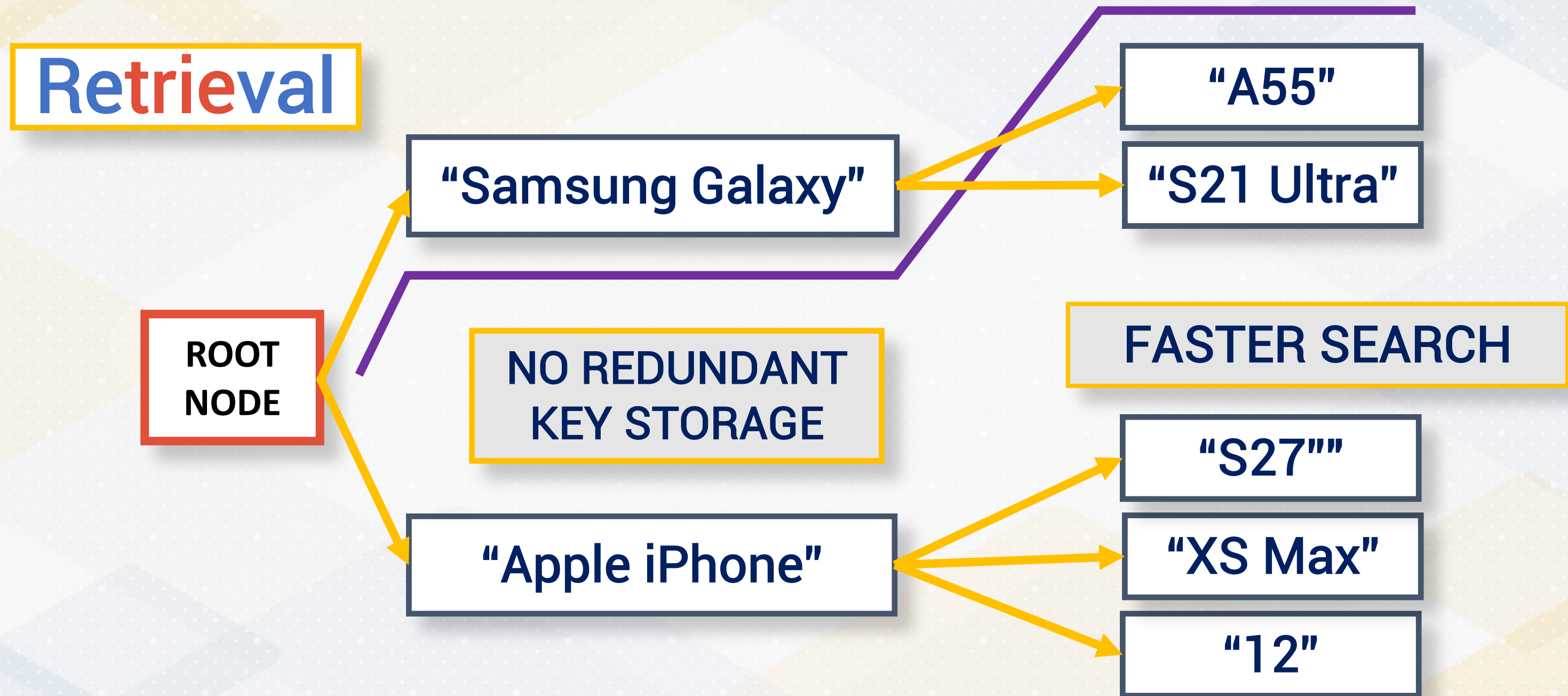
Product Name	Relevant or not
Samsung Galaxy S21 Ultra	Yes
Apple iPhone XS Max	No
Samsung Galaxy A55	Yes
Apple iPhone 12	No
Samsung Galaxy S27	Yes

```
// 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

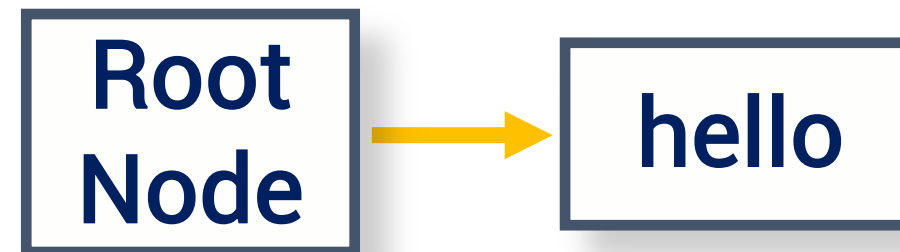
Whole Strings

Basic Trie

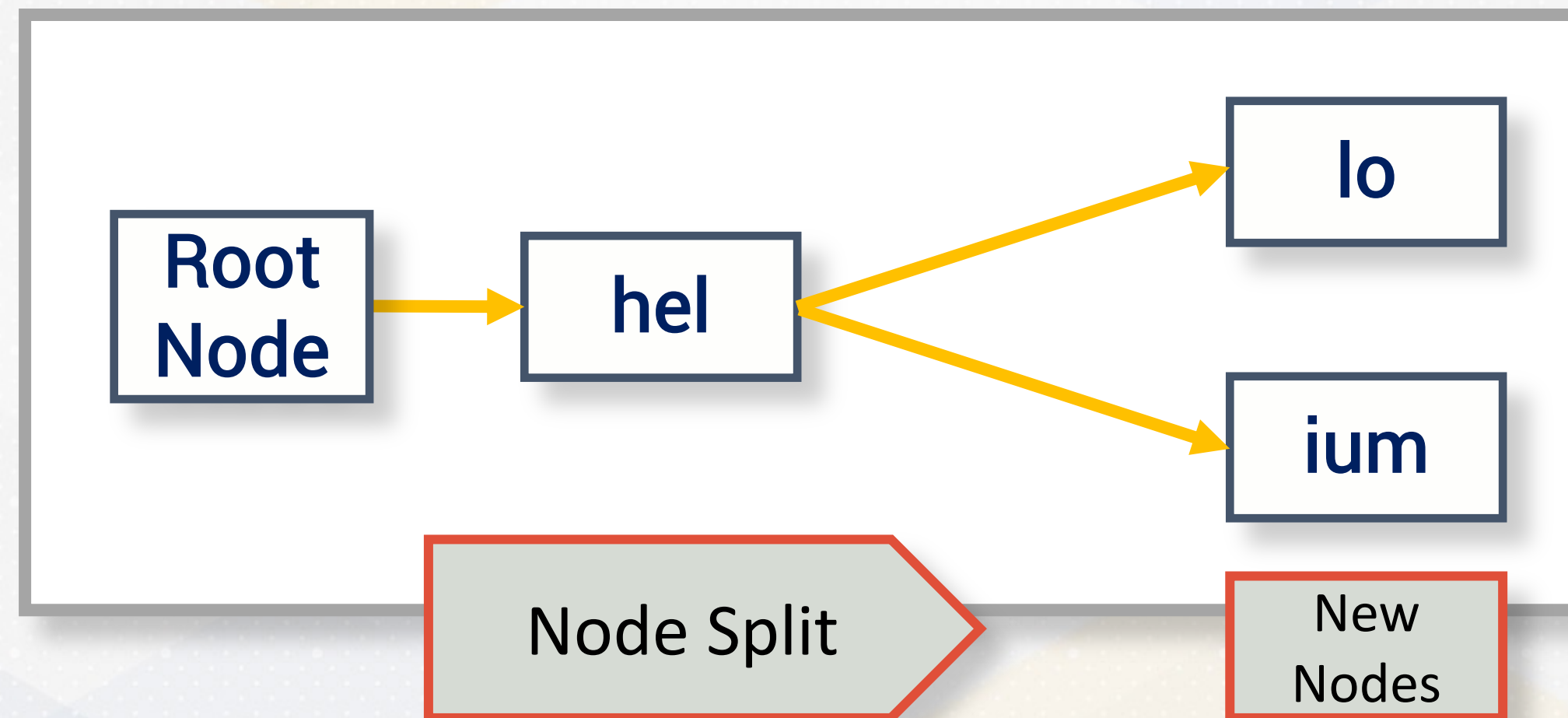
Patricia Trie



Insertion Example: “hello”



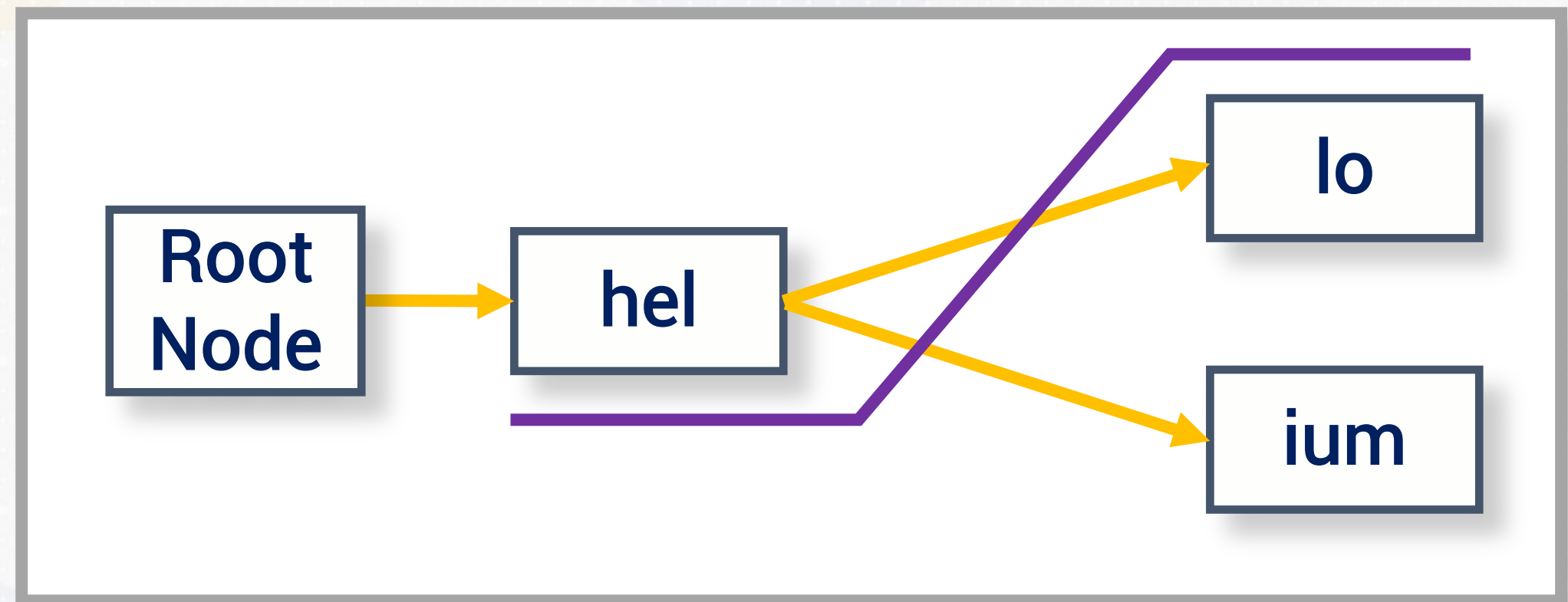
Insertion Example: “helium”



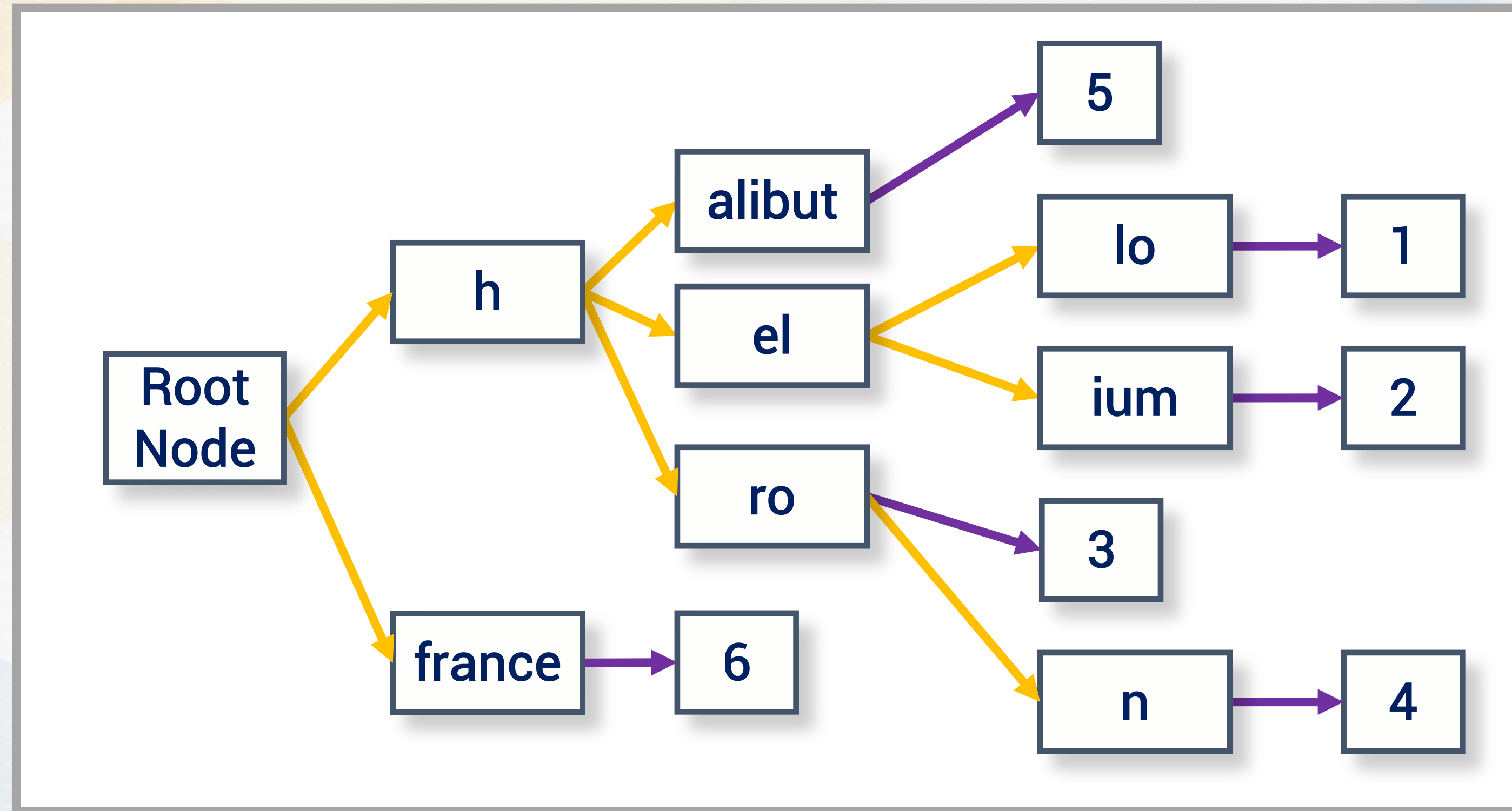
Retrieval: “hello” and “helicopter”

Navigate Root Node to
Relevant Leaf Node

Each Character
Narrows Down Search



More Complex Trie with Key-Value Pairs



Patricia Trie



Patricia Trie



```
class PatriciaNode {  
public:  
    bool isEndOfWord;  
    int value;  
    std::map<std::string, PatriciaNode *> children;  
  
    PatriciaNode() : isEndOfWord(false) {}  
};
```


Insertion

Termination Case

Stop Recursion

Find Substring

```
void insertHelper(PatriciaNode *node, const std::string &word, size_t index) {  
    // Termination 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

Common Prefix
Length

```
for (auto &child : node->children) {  
    // Calculate common prefix length  
    const std::string &key = child.first;  
    PatriciaNode *childNode = child.second;  
    size_t commonLength = 0;  
    while (commonLength < key.length() && commonLength < remaining.length() &&  
           key[commonLength] == remaining[commonLength]) {  
        commonLength++;  
    }  
    ...  
}
```


Insertion: Handling No Common Prefix

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

Node Splitting

```
if (commonLength > 0) {  
    if (commonLength < key.length()) {  
        // Split the child's key  
        std::string newKey = key.substr(0, commonLength); // "hel"  
        std::string oldKey = key.substr(commonLength);    // "lo"  
        PatriciaNode *newChildNode = new PatriciaNode();  
        newChildNode->children[oldKey] = childNode;  
        node->children[newKey] = newChildNode;  
        node->children.erase(key);  
        childNode = newChildNode;  
    }  
    // remainder: "ium"  
    insertHelper(childNode, word, index + commonLength, value);  
    return;  
}
```


Complexity Comparison of Trie and B+Tree

L: Average length of the key
N: Number of keys

	Patricia Trie	B+Tree
Insertion Time Complexity	$O(L)$	$O(\log N)$
Search Time Complexity	$O(L)$	$O(\log N)$
Space Complexity	$O(N * L)$	$O(N)$

Complexity Comparison of Trie and B+Tree

	Patricia Trie	B+Tree
Target Use Case	Fast prefix queries	Fast range queries
Memory Efficiency	Less efficient with dissimilar keys	More efficient, stores many keys per node
Range Queries	Supported but not efficient	Highly efficient

Inverted Index

Inverted Index

Map words to the positions in which they appear in a document.

Word	Positions
russia	10, 20
napolean	25

Proximity Search

distance("russia", "napoleon") less than ten words

Word	Positions
russia	10, 20
napoleon	25

Library Search

Map words to the documents in which they appear.

Word	Document ID, Position
russia	(1, 10), (1, 20)
napoleon	(1, 25), (2, 30), (3, 50)

Why Inverted Index?

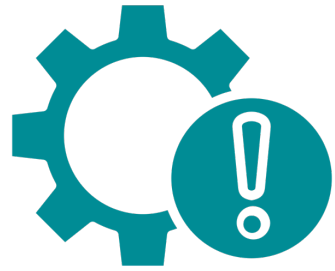
Inverted Index



Text
Data

Proximity
Search

Internals



Hash Table
Foundation

Values are
Doc IDs,
Positions

Structure of an Inverted Index

Key

"russia"

Value

{1: {5, 20}, 2: {10}}

```
std::unordered_map<  
    std::string,  
    std::unordered_map<int, std::vector<int>>  
>
```


Core Functions

Core Functions



addDocument



getDocuments



proximitySearch

Adding Documents to the Inverted Index

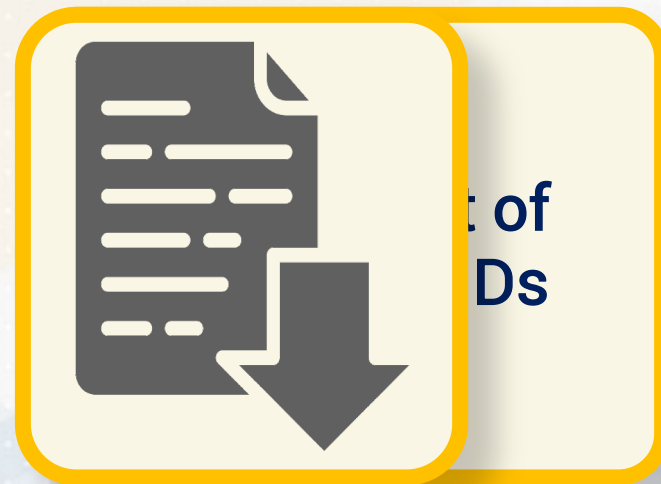


Add Word to
unordered_map

vert word to
lowercase if
needed

```
void addDocument(int docID, const std::string &content) {  
    documents.push_back(content);  
    std::vector<std::string> words = split(toLower(content));  
    for (size_t i = 0; i < words.size(); ++i) {  
        index[words[i]][docID].push_back(i);  
    }  
}
```

Retrieving Documents Containing a Term



```
std::unordered_set<int> getDocuments(  
    const std::string &word) {  
    std::string lowerWord = toLower(word);  
    if (index.find(lowerWord) != index.end()) {  
        std::unordered_set<int> docIDs;  
        for (const auto &entry : index[lowerWord]) {  
            docIDs.insert(entry.first);  
        }  
        return docIDs;  
    }  
    return {};  
}
```


Proximity Search: Finding Words Near Each Other

```
std::unordered_set<int> proximitySearch(const std::string &word1,
                                       const std::string &word2, int k)
{
    ...
    if (index.find(lowerWord1) != index.end() && index.find(lowerWord2) != index.end())
    {
        for (const auto &entry1 : index[lowerWord1]) {
            int docID = entry1.first;
            if (index[lowerWord2].find(docID) != index[lowerWord2].end()) {
                const auto &positions1 = entry1.second;
                const auto &positions2 = index[lowerWord2].at(docID);
                for (int pos1 : positions1) { for (int pos2 : positions2) {
                    if (std::abs(pos1 - pos2) <= k) { result.insert(docID); break; }
                }
            }
        }
        ...
    }
}
```

Benefits of Inverted Index



Efficient

Fast lookup for individual words



Complexity

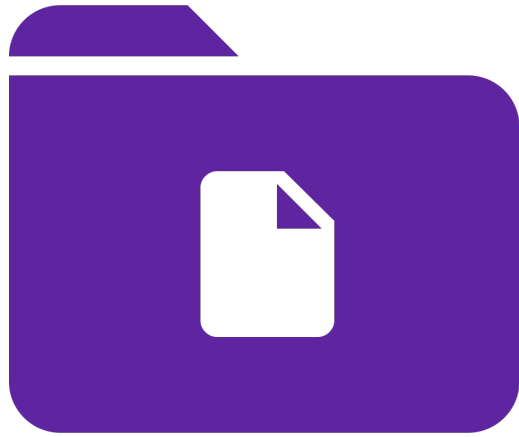
Proximity search and Boolean queries



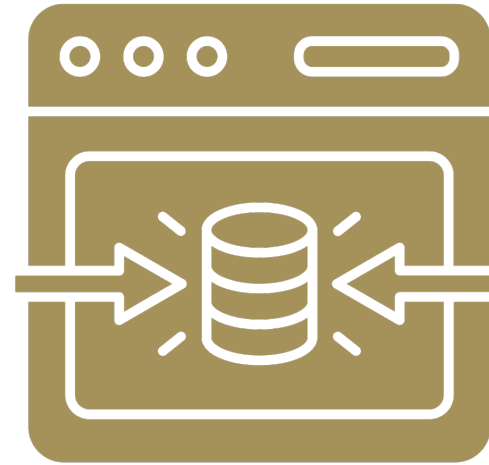
Scalable

Scales to large document collections

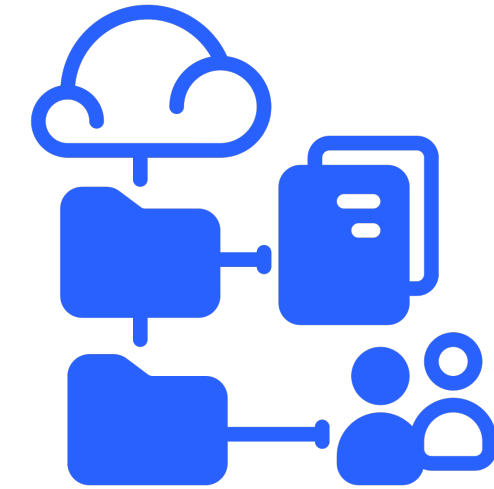
Challenges with Inverted Index



Large
Storage
Footprint



Compression:
Delta Encoding



Disk-Based
Storage
Management

Web-Scale Search



Stanford Digital Library Project (1995)



Héctor García-Molina



Sergey Brin Larry Page

Web-Scale Search



Distributed Inverted Index



Word-Based Sharding

Shards (subset of Words)	Replicas
Shard 1 (Words 1...100)	Servers A, B
Shard 2 (Words 101 ... 200)	Servers C, D
...	...

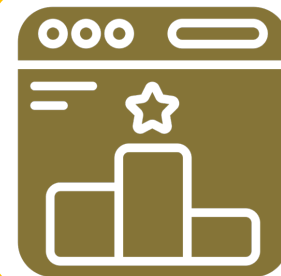
Document Ranking



**Relevant
Ranking**



**Ranking
Algorithm**



**Indexing
and Ranking**



Conclusion

- Trie
- Binary Patricia Trie
- Inverted Index
- Web-Scale Search