Database Course Assignment 1: Adaptive Radix Tree with Versioning

March 11, 2025

1 Introduction

In this assignment, you will enhance an existing ART implementation by incorporating features commonly required in modern database systems, particularly in the context of multi-version data management. These enhancements will enable ART to efficiently handle **range queries, multi-version storage, and historical data retrieval**. All operations in this assignment will be performed in a **single-threaded** environment.

2 Assignment Overview

We provide a basic ART implementation as a reference. You may reconstruct or rewrite the code in your preferred style. In this assignment, the value associated with each key represents a **row ID**, which corresponds to a position in a large table stored separately. Your goal is to modify the ART to support the following functionalities:

1. Implement a function for range scan.
2. Extend the ART to support multi-version value storage.
3. Optimize query performance by replacing linked lists with skip lists.
4. (Bonus) Analyze the impact of versioning on range query locality and propose potential optimizations.
5. Integrate your ART implementation into BabyDB.

3 Task 1: Implement a Function for Range Scan.

To efficiently support range queries.

- `ScanRange(key1, key2)`: Retrieves all key-value pairs where the key falls within the closed interval $[key1, key2]$.
- The complexity of the range scan should be $O(H + L)$ H is the hight of the tree and L is the scan length.

4 Task 2: Multi-Version Value Storage

Modern databases require the ability to maintain multiple versions of a key's value over time. Instead of overwriting an existing value, historical versions must be preserved. This allows:

- **Historical queries**: Users can retrieve past versions of data.
- **Multi-Version Concurrency Control (MVCC)**: A mechanism that allows concurrent transactions to access different versions of the same data. Although MVCC will be covered in a subsequent assignment, this task focuses solely on multi-version storage.

To implement this feature:

- Modify the ART `insert` function to include an additional **timestamp** parameter, indicating when the key-value pair was inserted.
- Each slot in a leaf node should store:
  - The key.
  - The timestamp at which the value was inserted.
  - A pointer to the head of a linked list, which stored the oldest value of the key. Newer values of the key are stored in the linked list.
- If a key already exists, instead of replacing the old value, append a new node to a linked list. This linked list should maintain:
  - The `row IDs` associated with the key.
  - The timestamp when the a row ID was inserted.
  - A pointer to the newer version (if any).
- The linked list for each key should be sorted in increasing order based on timestamps. However, **timestamps are not guaranteed to be inserted in increasing order**. By default, the initial timestamp of a value is 100.
- The validity of each version extends from its insertion timestamp until a newer version (with a larger timestamp) is added.

Query Modifications: Modify point and range queries to allow filtering by **time**. Queries should return only the key-value pairs that are valid at a specified time.

5 Task 3: Replacing Linked Lists with Skip Lists

Storing multiple versions in a linked list leads to inefficient queries, as linked list traversal is slow. To address this, replace the linked list with a **skip list**, which offers logarithmic-time complexity for search operations.

Skip List Implementation:

- Modify the ART to store historical versions using a **skip list** instead of a linked list.
- Ensure that the skip list maintains elements in increasing timestamp order.
- Modify query functions to leverage the skip list structure for efficient historical value retrieval.

Reference: For more information on skip lists, refer to: `https://en.wikipedia.org/wiki/Skip_list`

6 Task 4 (Bonus): Analyzing the Impact of Versioning on Range Query Locality

Range queries in an ART benefit from the tree's structure, which provides spatial locality by storing nearby keys within the same node. However, when multiple versions of a key's value are stored in a linked list or skip list, this locality can degrade.

Analysis:

- Explain why scanning linked lists or skip lists negatively affects locality during range queries.
- Provide an example scenario illustrating this issue.
- Suggest potential solutions to mitigate the loss of locality. (Implementation is not required, but additional points will be awarded for implemented solutions.)

7 Task 5: Integrating ART with BabyDB

Once you have implemented and optimized the ART, integrate it into BabyDB, a simple database system.

Integration Requirements:

- Your ART should function as an alternative index structure in BabyDB.
- The API design within ART is flexible; however, BabyDB will only call the indexing APIs defined in `Index.hpp`.

8 Grading

To evaluate your implementation and understanding of Task 4, you are required to submit a report that follows a structured and academic format. The report should thoroughly explain your interpretation of the **ART** design logic and provide a detailed explanation of your **range query** implementation. Specifically, you should address the following aspects:

- **Design Logic of ART (Adaptive Radix Tree):**
  - Explain the core design principles of ART and its motivation.
  - Discuss how ART improves upon traditional radix trees and Trie.
  - Explain how ART supports efficient key insertions, deletions, and lookups.
- **Implementation of Range Query:**
  - Describe the algorithm used to implement the range query operation in ART.
  - Explain how the traversal mechanism works to efficiently retrieve all keys within a given range.

The grading will be based on the following components:

- **Correctness of Implementation (9 points)**
  - Your code will be tested against **12 test cases**, each worth **0.75 points**, for a total of **9 points**.
- **Report Quality (3 points)**
  - The clarity and coherence of your explanation.
  - A well-structured report that logically presents your understanding.
  - Proper use of academic writing style.
- **Code Readability (3 points)**
  - Clear, well-commented code.
  - Efficient and structured code that is easy to understand.
- **Bonus (Up to 5 points)**