Lecture 18: Learned Index

Logistics

- Programming assignment 3 (B+Tree) due on **Nov 2**

Recap

- RTree
- ND RTree

Lecture Overview

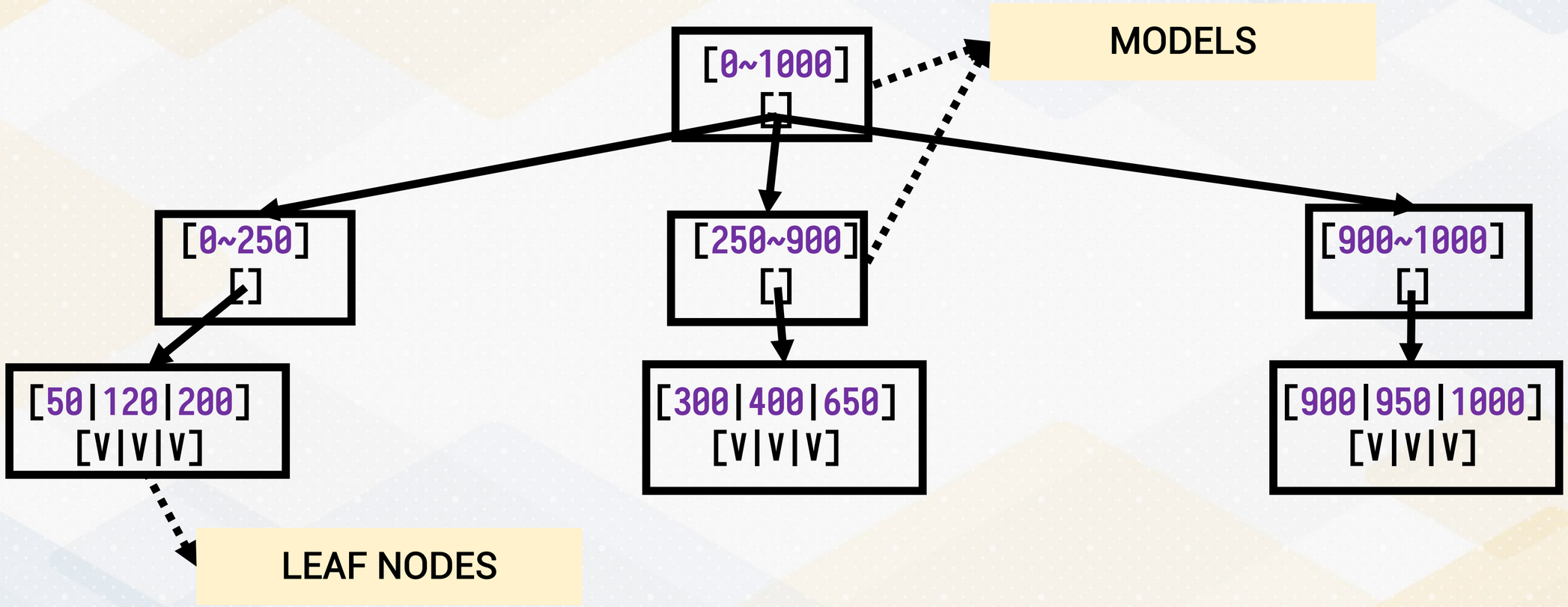
- Learned Index
- Learned Index using Neural Network

Learned Index

Limitations of B+Tree

- **Fixed Structure**:
  - Hierarchical nodes require multiple levels of traversal (log(n) comparisons), leading to higher CPU and memory costs.
- **Memory Overhead**:
  - Each node holds pointers, using significant memory for large datasets.
- **Rigid Partitioning**:
  - B+ Trees don't adapt to predictable key patterns, which can cause unnecessary disk I/O and slower access.

Learned Index



Learned Index

- **Data-Adaptive Structure**:
  - **Root Model** predicts which region (sub-model) to access, based on the data's distribution.
  - **Sub-models** provide fine-grained predictions, reducing the need for multiple tree levels.
- **Reduced Memory and Faster Lookups**:
  - Learned indexes minimize the need for pointers and log(n) comparisons, using a more compact model.

History

- In **2018**, Google researchers proposed the **learned index** as an innovative approach to indexing.
- Inspired by advancements in machine learning, they recognized that indexes could be trained as models to predict data positions instead of relying on rigid structures.

Tim Kraska (2018)

Key Idea

- **Data Distribution Awareness**: Learned indexes use machine learning (often regression models) to approximate the mapping of keys to positions, adapting to the underlying data distribution.
- **Predictive Power**: By leveraging data patterns, a learned index can skip multiple search steps typical in traditional indexes.

B+Tree vs Learned Index

	B+ Tree	Learned Index
Structure	Fixed, hierarchical	Data-adaptive, predictive
Search Cost	Log(n) comparisons	Direct prediction with neighborhood search
Memory Use	High, with pointers per node	Lower, fewer nodes needed
Adaptability	Fixed partitions, no data patterns	Fits patterns, reduces overhead

Structure of a Learned Index

- **Root Model**: Predicts a range of data positions.
- **Sub-models**: Finer-grained models handle local data within that range.

```
struct Model {
    double slope = 0.0;
    double intercept = 0.0;
    void train(const std::vector<int>& keys, const std::vector<size_t>& positions);
    size_t predict(int key) const;
    void print() const;
};
```

Training the Root and Sub-models

- **Root Model**: Learns a rough mapping of keys to sub-models.
- **Sub-models**: Fine-tune predictions for localized clusters of data.

```
void train(const std::vector<int>& keys, const std::vector<size_t>& positions) {
    // Calculate mean
    for (size_t i = 0; i < keys.size(); ++i) {
        mean_x += keys[i];
        mean_y += positions[i];
    }
    // Compute slope and intercept
    slope = num / denom;
    intercept = mean_y - slope * mean_x;
}
```

Regression

- A statistical method for modeling the relationship between a dependent variable (x) and an independent variable (y).

$$y(x) = slope * x + intercept$$

$$Position(key) = slope * key + intercept$$

Training Regression Model

- Calculate the **mean** of keys (x) and positions (y).
- Find the **slope** and **intercept** to define the linear relationship.

position	0	1	2	3	4
key	50	100	120	200	250

Training Regression Model

- Trains models by calculating mean, slope, and intercept to fit the data distribution.

```
 double num = 0.0, denom = 0.0;
    for (size_t i = 0; i < keys.size(); ++i) {
       double x_diff = keys[i] - mean_x;
       num += x_diff * (positions[i] - mean_y);
       denom += x_diff * x_diff;
    }
}
slope = num / denom;
intercept = mean_y - slope * mean_x;
```

Training Regression Model

- Numerator accumulate values to compute the covariance of keys and positions.
- Denominator accumulates the variance of keys

```
slope = num / denom = 500 / 20000 = 0.025

intercept = mean_y - (slope × mean_x) = 2 - (0.025 × 150) = -1.75

y = 0.025 × x - 1.75
```

Trained Regression Model

$$y = 0.025 \times x - 1.75$$

position	0	1	2	3	4
key	50	100	120	200	250
y	-0.5	0.75	2	3.25	4.5

Loading Training Data

- Root and sub-models are trained sequentially, with each sub-model learning its local cluster of data.

```
for (int i = 0; i < num_sub_models; ++i) {
    cluster_keys.clear();
    cluster_positions.clear();
    for (size_t j = start; j < end; ++j) {
        cluster_keys.push_back(sorted_data[j].first);
        cluster_positions.push_back(j);
        leaves[i].push_back(sorted_data[j]);
    }
    sub_models[i].train(cluster_keys, cluster_positions);
}
```

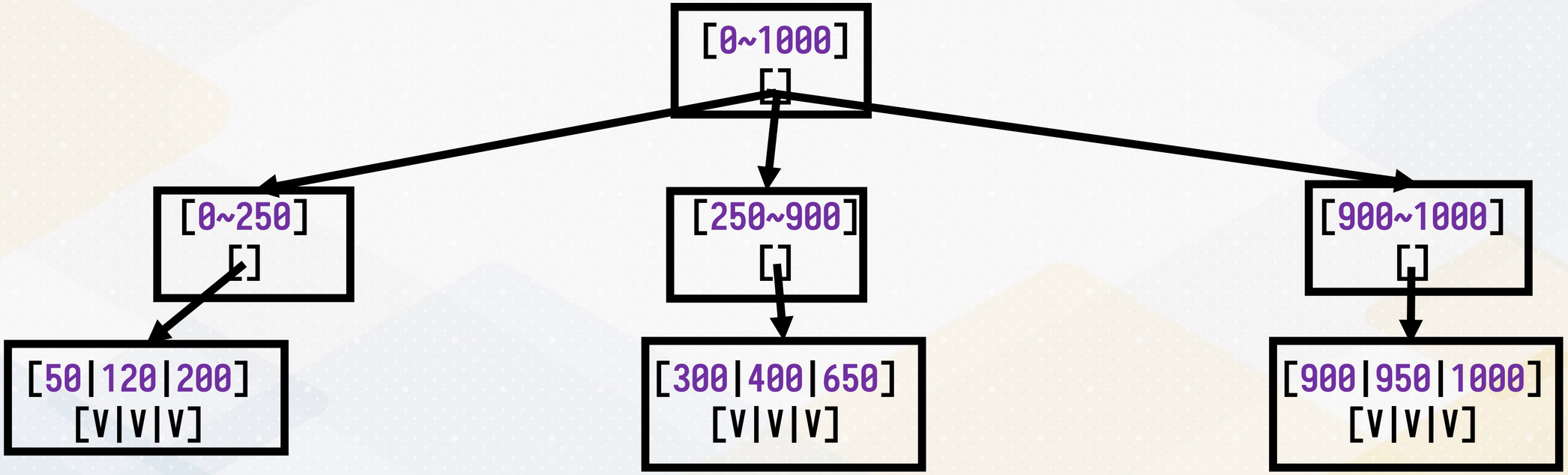
Position Prediction Process

- Root model predicts the sub-model to use.
- Sub-model predicts the approximate position of the key within the data cluster.

```
int sub_model_idx = root_model.predict(key);
size_t pos = sub_models[sub_model_idx].predict(key);
```

Position Prediction Process

- Root model predicts the sub-model to use.
- Sub-model predicts the approximate position of the key within the data cluster.



Using Trained Regression Model

- Map key to position

```
size_t predict(int key) const {
    return static_cast<size_t>(std::max(0.0, slope * key + intercept));
}
```

Executing a Search with Learned Index

- Root model predicts sub-model.
- Sub-model predicts approximate position within the data cluster.

```
int sub_model_idx = root_model.predict(key);
size_t pos = sub_models[sub_model_idx].predict(key);
```

Neighborhood Search Around Predicted Position

- Searches nearby positions for accuracy, as regression may only predict an approximate location.

```
int search_radius = 5;
int start = std::max(0, int(pos) - search_radius);
int end = std::min(int(leaves[sub_model_idx].size() - 1), int(pos) + search_radius);

for (int i = start; i < end; ++i) {
    if (key == leaves[sub_model_idx][i].first) {
        return leaves[sub_model_idx][i].second;
    }
}
```

Visual Example of Neighborhood Search

- Predicted position = 10
- Search radius = 2

8	9	10	11	12
3	7	15	25	32

Practical Considerations for Neighborhood Search

- **Choosing Radius**:
- Larger radius increases the likelihood of finding a key but will reduce speed.
- Radius depends on data distribution and prediction accuracy of the regression model.
- **Optimizing for Real-World Data**:
- Skewed or clustered data may benefit from dynamic radius adjustment.

Testing the Learned Index

- Demonstrates search for keys within predicted ranges, highlighting the search speedup over traditional methods.

```
for(auto search_key : search_keys){
    std::cout << "Search for key " << search_key << " : " <<
    index.search(search_key) << std::endl;
}
```

Benefits and Limitations of Learned Indexes

- **Benefits**:
- Fast lookups with reduced memory usage.
- Flexible to data distributions.
- **Limitations**:
- May need neighborhood search due to approximate predictions.
- Effective for clustered data; less accurate for highly scattered data.

Evaluation Metrics

- **Lookup speed:** Learned indexes can transform lookup operations from the typical B-Tree complexity of O(log n) into near-constant time operations.
- **Memory usage:** Learned indexes use less memory by eliminating redundant pointers and nodes.
- **Prediction accuracy:** By training models to understand the data's cumulative distribution function (CDF), learned indexes achieve high accuracy for point and range queries, even with relatively simple models like linear regression.

Learned Index using Neural Network

Limitation of Learned Index using Regression

- The relationship between keys and positions can be highly non-linear, with gaps increasing at an exponential rate.
- A linear regression model would struggle to capture these intervals accurately.

0	1	2	3	4	5	6	7	8
10	20	45	100	200	350	600	1000	1500

Non-Linearity

- **Data Distribution**: Real-world datasets often exhibit complex, non-linear relationships that cannot be accurately modeled by a simple line or function.
- **Examples**: Customer purchase behavior, seasonal data patterns, and geographic data often show non-linear trends.
- **Limitations of Linear Models**: Linear regression can only capture straightforward relationships.
- For data that fluctuates unpredictably or has clusters and non-uniform distributions, linear models fall short.

Neural Network

- A neural network can capture non-linear data patterns by learning the complex mapping.
- With multiple hidden layers and neurons, they can model intricate relationships that simpler models miss.

Abstract Model Class

- Defines a general model interface with train, predict, and print methods.

```
struct Model {
    virtual ~Model() {}
    virtual void train(const std::vector<int>& inputs, const std::vector<double>& targets) = 0;
    virtual double predict(int x) const = 0;
    virtual void print() const = 0;
};
```

SimpleNeuralNetwork Class

- A single-layer neural network model with input_weights, output_weights, input_bias, and output_bias.
- hidden_neurons: Defines the number of neurons in the hidden layer.

```
class SimpleNeuralNetwork : public Model  {
    std::vector<double> input_weights;
    std::vector<double> output_weights;
    double input_bias, output_bias;
    int hidden_neurons;
    // other variables and helper functions...
};
```

Network Structure

- **Input Layer**: 1 neuron for the key.
- **Hidden Layer**: 10 neurons, with weights and biases initialized randomly.
- **Output Layer**: 1 neuron for the predicted position.

```
SimpleNeuralNetwork nn(10); // 10 hidden neurons
```

Training the Neural Network for Learned Index

- **Data Normalization**:
  - Inputs are normalized to [0, 1] to enhance training performance.

```
void train(const std::vector<int>& inputs, const std::vector<double>& targets) {
    for (int epoch = 0; epoch < epochs; ++epoch) {
        for (size_t i = 0; i < inputs.size(); ++i) {
            // Forward and backward pass...
            output_bias += learning_rate * delta_output;
            input_bias += learning_rate * delta_hidden;
        }
    }
}
```

Training the Neural Network for Learned Index

- **Forward Pass:** Calculates hidden layer outputs using sigmoid activation, then sums outputs for final prediction.
- **Backward Pass:** Adjusts weights and biases based on error between prediction and actual target.
- **Epochs and Learning Rate**: Trains over multiple epochs with a small learning rate to improve prediction accuracy.

Training the Neural Network for Learned Index

- **Input Normalization**:
- Normalize each key by dividing by the maximum key (1500) to improve training.
- **Input to Hidden Layer**:
- Each hidden neuron computes its output using:

$$h_j = \sigma(x \cdot w_{i,j} + b_j)$$

- Here, x is the normalized key, w_i,j is the weight for hidden neuron j, and b_j is the bias for neuron j.

Training the Neural Network for Learned Index

- **Hidden to Output Layer**:
- Compute the final predicted position by summing up:

$$y = \sum (h_j \cdot w_{h,j}) + b_o$$

- Here, h_j: Output of hidden neuron j; w_hj: weight from hidden neuron j to output; b_o: output bias.

$$\text{Loss} = \frac{1}{n} \sum (\text{predicted position} - \text{actual position})^2$$

Example Iteration

- (key = 10, position = 0)
- Key is normalized to 10/1500 = 0.0067
- Assume initial weights and biases are small values, e.g., 0.1, 0.1.
- h_1=sigmoid(0.0067·0.1+0.1) ≈ 0.525
- Repeat for each hidden neuron
- Compute predicted position y
- Suppose the predicted position for key = 10 is initially 0.5
- Calculate the loss: (0.5−0)2=0.25(0.5 - 0)^2 = 0.25(0.5−0)2=0.25.
- Perform backpropagation to adjust weights and biases for improved predictions on the next pass.

Recursive Model Index

- Root and Sub-Models in the RMI Using Neural Networks

```
void initModels(bool use_neural_network) {
    if (use_neural_network) {
        root_model = std::make_unique<SimpleNeuralNetwork>();
        for (int i = 0; i < 4; ++i) {
            sub_models.push_back(std::make_unique<SimpleNeuralNetwork>());
        }
    }
}
```

Using Neural Network Prediction for Key Search

- The root_model predicts which sub-model to use.
- The selected sub-model then predicts the position of the key within its data cluster.

```
int sub_model_idx = root_model->predict(key);
size_t pos = sub_models[sub_model_idx]->predict(key);
```

Benefits and Limitations

- **Benefits**:
- Flexibility in capturing non-linear data patterns.
- Reduced lookup time by direct key position prediction.
- **Considerations**:
- Requires tuning for hidden neuron count, learning rate, and epochs.
- Higher memory usage due to network weights.

Conclusion

- Learned Index
- Learned Index using Neural Network