Implementing a CSV Index with a B-Tree: Takeaways

Syntax

- CSV index implementation:

```
class CSVIndex(BTree):
    def __init__(self, split_threshold, csv_filename, col_name):
        super().__init__(split_threshold)
        self.col_name = col_name
        with open(csv_filename) as file:
            rows = list(csv.reader(file))
            header = rows[0]
            rows = rows[1:]
            col_index = header.index(col_name)
            for row in rows:
                self.add(float(row[col_index]), row)
    def _range_query(self, range_start, range_end, current_node, min_key, max_key):
        if range_start > max_key or range_end < min_key:
            return []
        results = []
        for i, key in enumerate(current_node.keys):
            if range_start <= key and key <= range_end:
                results.append(current_node.values[i])
        if not current_node.is_leaf():
            for i, child in enumerate(current_node.children):
                new_min_key = current_node.keys[i - 1] if i > 0 else min_key
                new_max_key = current_node.keys[i] if i < len(current_node) else max_key
                results += self._range_query(range_start, range_end, child, new_min_key,
new_max_key)
        return results
    def range_query(self, range_start, range_end):
        return self._range_query(range_start, range_end, self.root, float('-inf'),
float('inf'))
    def save(self, filename):
        with open('{}.pickle'.format(filename), 'wb') as f:
            pickle.dump(self, f)
    @staticmethod
    def load(filename):
        with open('{}.pickle'.format(filename), 'rb') as f:
            return pickle.load(f)
```

Concepts

- A B-tree stores unique keys. When we use non-unique keys, we lose the performance guarantees for range queries. More specifically, with unique keys, we're sure that a range query will never visit a subtree that doesn't contain query results.
- With unique keys, the time complexity of a range query is $O(min(n, r \times log(n)))$, where r is the number of query results, and n the height of the tree.
- The split threshold is important. Increasing it makes processing each node slower, but it also reduces the number of nodes. Finding the right balance is key for high-performance indexes.
- B-trees are particularly suited for storage on disk. Having large nodes reduces the number of disk reads but slows down the search.
- The time complexity of building a B-tree with n nodes is $O(n \log(n))$.

Resources

- Postgres B-tree blog post
- Postgres B-tree documentation
- B+ trees
- pickle module