Design Document: Blocked Sequence Set Zip Code Processing System

Project Overview:

1. Storage of variable-length records in fixed-size blocks (512 Bytes)
2. Indexing and searching Zip Code records using primary keys and a simple index
3. Readable dump methods for physical and logical views (RBN)
4. Command-line options to use the program

Architecture

1. Buffer implementation

The following outlines the architecture and implementation of the buffer system as it processes and organizes large datasets of ZIP Code records. The system is designed around a std::unordered_map (hashmap) of hashmaps to efficiently handle the data, with a primary focus on scalability, modularity, and ease of data processing.

Purpose

The purpose of this system is to:

1. **Read** and parse a CSV file containing ZIP Code information.
2. **Organize** the data into memory-efficient blocks for optimized access.
3. **Enable** block-wise and global operations like sorting, searching, and debugging.

The chosen data structure allows fast lookups, dynamic updates, and clear separation of concerns.

System Design

Core Data Structure

- **Primary Structure:** std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord>>
    - **Outer HashMap** (std::unordered_map<size_t, ...>):
        - **Key**: size_t representing the block number (0-based index).
        - **Value**: Another std::unordered_map representing the ZIP Code records within that block.
    - **Inner HashMap** (std::unordered_map<std::string, ZipCodeRecord>):

        - **Key**: std::string representing the ZIP Code.
        - **Value**: A ZipCodeRecord struct containing details of the ZIP Code.

Example of Data Organization

Block Number	Inner HashMap (ZIP Code Records)
0	{ "12345": ZipCodeRecord, "67890": ZipCodeRecord }
1	{ "54321": ZipCodeRecord, "98765": ZipCodeRecord }

Classes and Responsibilities

1. **ZipCodeRecord (Data Structure)**
    a. A lightweight structure holding individual ZIP Code data.
    b. Fields:
        i. zip_code (string): ZIP Code.
        ii. city (string): Name of the city.
        iii. state_id (string): State abbreviation (e.g., "MN").
        iv. latitude (double): Latitude of the ZIP Code location.
        v. longitude (double): Longitude of the ZIP Code location.
2. **BlockBuffer (Helper Class)**
    a. Responsible for managing a single block of records (one inner hashmap).
    b. Provides functionality to:
        i. Convert the block into a std::vector<ZipCodeRecord> for sequential access.
3. **RecordBuffer (Helper Class)**
    a. Manages a single record for processing.
    b. Provides functionality to:
        i. Extract and store fields of a ZipCodeRecord.
        ii. Output the record details for debugging or logging.
4. **Buffer (Core Class)**
    a. The central class that integrates the system.
    b. Responsibilities:
        i. **Data Loading**: Reads a CSV file and populates the hashmaps.
        ii. **Data Storage**: Organizes records into blocks based on a predefined block size.
        iii. **Data Processing**: Allows operations like sorting, block-wise processing, and dumping of records.
    c. **Key Methods**:
        i. read_csv: Parses a CSV and stores records in the hashmap-of-hashmaps structure.
        ii. process_blocks: Iterates through blocks to unpack and process records.
        iii. sort_records: Sorts all records in memory by ZIP Code.
        iv. dump_blocks: Debugging utility to print all blocks and their contents.

Design Decisions

1. **HashMap of HashMaps Design**
   a. **Reasoning**: We decided to choose the hierarchical hashmap structure for the following reasons:
      i. **Scalability**: Each block (outer hashmap) can handle thousands of records independently.
      ii. **Efficiency**: Lookups by block number or ZIP Code are O(1) on average due to the properties of hashmaps.
      iii. **Modularity**: Clear separation between block-level and record-level operations.
2. **Dynamic Block Creation**
   a. Blocks are created dynamically based on the number of records read.
   b. Records are evenly distributed across blocks, determined by a configurable block size.
3. **Modular Design**
   a. Separate helper classes (BlockBuffer and RecordBuffer) ensure that block-level and record-level processing are encapsulated and reusable.

2. Header Record Buffer:

The Header Record component manages metadata for the blocked sequence set files, storing information about file organization, block structure, and field definitions required for proper data processing.

Header Record Structure

File Structure Information:

- File structure type
- Version number
- Header size
- Record size format
- Size format type (ASCII/binary)

Block Information:

- Block size (512 bytes default)
- Minimum block capacity (50% default)
- Block and record counts

Field and Index Information:

- Index file name and schema
- Primary key field
- Field definitions (name, type schema)

Chain Management:

- Available list RBN
- Active list RBN
- Stale flag

Header Record Implementation

Format:

- Length-indicated fields: [LL]data,
- LL: Two-digit length
- Comma-separated values

Key Components:

- HeaderRecord class: Reads/writes header information
- FieldMetadata struct: Stores field definitions

Operations:

- Write header in length-indicated format
- Read and parse header
- Maintain metadata and field definitions
- Update RBN links

Blocking Sequence Set file:

- The block sequence set file is formatted as so, RBN,record1,...,record
- This is followed for every block with a max size of 512 bytes
- If a record would put the current block over 512 bytes it will be put in the next block to not have partial records in blocks.

Block Sequence Set/Record Buffer

Index File:

- Description:

- An index file mapping primary keys (Zip Codes) to their Relative Block Numbers (RBN) and positions within blocks.
- Implements search functionality to retrieve records using the index.
- Ensure efficient and accurate retrieval without scanning the entire file.

- Index File Structure:
    - Key-Value Pairs: <Primary Key, RBN>
    - If records span blocks, include position offsets for precise retrieval.

- Implementation:
    - During record insertion into blocks, simultaneously update the index.
    - Sort the index by primary keys for faster searches.
    - Store the index in a separate binary file.

Search Functionality for dump in main:

- Command-Line Search:
    - Users can search records by primary key.
    - Example syntax: search <ZipCode>

- Search Algorithm:
    - Look up the primary key in the index.
    - Fetch the corresponding RBN.
    - Read the block containing the record and extract it.

- Index Dump:
    - Syntax: dump -index
    - Displays the current index for debugging and verification.