IN6226 Information Retrieval Analysis / Assignment 2 / AY24-25

Name	Matric	Email
HUANG YIJING	G2405655L	YIJING005@e.ntu.edu.sg
LIANG HOU	G2401631D	HLIANG008@e.ntu.edu.sg
XIE QINGLING	G2405624J	QINGLING001@e.ntu.edu.sg
ZENG HUA	G2405796B	HUA002@e.ntu.edu.sg

1. Inverted Index Construction

Step 1: Input and Motivation

Given that we used BSBI in Assignment 1, we started from the output: sorted term-document pairs stored in multiple block_*.json files. These were merged into a global sorted list using a heap-based multiway merge algorithm.

Step 2: Building the Inverted Index

We aggregated the sorted pairs into an inverted index using a Python dictionary, where each term maps to a list of document IDs. Document IDs are stored as sorted, deduplicated integers.

The resulting index was saved as final_index.json for further processing. This structure supports fast retrieval and is easy to compress.

2. Boolean Retrieval Implementation

To support Boolean search over our inverted index, we implemented two query processors:

- BooleanQueryProcessor: Works on the original uncompressed inverted index stored as a JSON file.
- CompressedBooleanQueryProcessor: Works on the compressed index with delta and variable-byte encoding.

These two processors share the same interface and are used in run_query_examples() to compare their behavior and performance.

BooleanQueryProcessor

This class supports basic Boolean queries using the uncompressed index. It scans the query from left to right and applies AND, OR, and NOT operators in sequence. It uses Python sets to compute intersections, unions, and differences. However, it does not consider operator precedence—queries are evaluated in the order they appear.

```
if current_op == 'AND':
```

```
        result_set &= doc_set
    elif current_op == 'OR':
        result_set |= doc_set
    elif current_op == 'NOT':
        result_set -= doc_set
```

CompressedBooleanQueryProcessor

This class is designed for querying the compressed index. It supports the same Boolean operators, but with correct operator precedence: NOT > AND > OR.

It uses two stacks to evaluate expressions: one stack holds operands (posting lists), one stack holds operators.

During parsing, if a NOT operator is found, the next term is immediately processed and decompressed on demand. This is implemented in the following logic:

```
if current_op == 'NOT':
    next_term = terms[i+1].lower()
    term_id = self.compressed_index.term_to_id.get(next_term, -1)
    if term_id == -1:
        doc_set = set()
    else:
        doc_set = set(self.compressed_index.decompress_term(term_id))
    operand_stack.append(doc_set)
    i += 2
```

Only when a term appears in the query do we decompress its posting list, which avoids unnecessary memory usage. After parsing all terms and operators, the stacks are evaluated respecting operator precedence.

3. Index Compression and Optimization

To reduce the memory footprint of the final inverted index, we implemented a two-stage compression strategy: Delta Encoding followed by Variable-Byte Encoding. This technique allows efficient storage and retrieval of postings lists while maintaining query functionality.

Compression Strategy

The compression logic is encapsulated in the CompressedIndex class, which processes the final inverted index and transforms each postings list (i.e., list of document IDs) into a compact binary format. The compression steps are:

1.Term Dictionary Mapping:

Each term is assigned a unique integer ID to avoid storing long string tokens repeatedly. This is achieved using the method:

```
def build_term_dictionary(self):
    sorted_terms = sorted(self.original_index.keys())
    for term_id, term in enumerate(sorted_terms):
        self.term_to_id[term] = term_id
        self.id_to_term[term_id] = term
```

2. Delta Encoding:

Since document IDs in a postings list are strictly increasing, we store the difference (delta) between consecutive values. For example:

[100, 104, 107] → [100, 4, 3].

3. Variable-Byte Encoding (VB):

The delta values are then encoded into a variable number of bytes. VB encoding uses the most significant bit (MSB) of each byte as a continuation flag. The final byte of each number is marked by setting its MSB to 1.

```
    def variable_byte_decode(self, encoded_bytes):
        numbers = []
        current_num = 0
        for byte in encoded_bytes:
            if byte & 0x80:
                current_num = (current_num << 7) | (byte & 0x7F)
                numbers.append(current_num)
                current_num = 0
            else:
                current_num = (current_num << 7) | byte
```

```
        return numbers
```

4. Storage Format:

The encoded byte arrays are stored in a binary file (.compressed), along with a dictionary file (.term_dict) that maps term IDs to strings and vice versa.

Decompression on Demand

When a Boolean query is executed (via CompressedBooleanQueryProcessor), decompression is performed only for the terms involved in the query. This keeps memory usage minimal.

Results

The compression yields a 42.31% reduction in space compared to the original JSON index:

```
Original index size: 3072.05 KB

Compressed index size: 1772.34 KB

Compression ratio: 57.69%

Space savings: 42.31%
```

4. Query Testing & Performance

We compared the compressed and uncompressed index performance on a set of sample queries:

Query	Docs Returned	Uncompressed (ms)	Compressed (ms)
clinton AND email	119	0.13	0.32
obama OR president	377	0.06	0.13
hillary NOT bill	0	0.03	0.08
security AND (white OR house)	0	0.01	0.01
nonexistent_term	0	0.00	0.00

Each query was timed, and the document count returned was recorded. The results were consistent across both index versions, confirming correctness.

We observed that while the compressed version had slightly longer query times, the reduction in index size is substantial and worthwhile.

5. Conclusion

We successfully implemented a scalable indexing system with compression and Boolean search. The compressed index achieved a 42% reduction in size while maintaining accurate and fast query responses.