

suppose we are encoding all our documents by converting each token into hash - 64 bit integer. we want to keep references to documents for each encoded token. we are using tokenizer that consistently extract the same hash for the same words in the documents.

we also using a compact presentation of each hash by converting hash into pair  $(b, z)$ , where  $b$  is a bucket number that calculated as an integer generated from  $k$  leading bits of the hash integer, and  $z$  is a number of running zeros from the end of the hash

2 task:

1. efficient persistence of token hashes and doc references (inverted index),  $(b, z)$  should be a part of inverted index;
2. algorithm using constraint programming that allows us to restore document tokens using only  $(b, z)$  associated with given document. we have estimation of number of distinct tokens in the document (cardinality). we can use this cardinality as constraint.



## Task 1: Efficient Persistence of Token Hashes and Document References (Inverted Index)

To efficiently store token hashes and document references, we can use an **inverted index** structure. The inverted index will map each token hash (represented as  $(b, z)$ ) to a list of documents containing that token. Here's how we can design it:

Data Structure:

1. **Bucket (  $b$  )**: The bucket number is derived from the leading  $k$  bits of the 64-bit hash. This reduces the number of unique keys in the inverted index, making it more compact.
2. **Zeros (  $z$  )**: The number of trailing zeros in the hash. This helps in further compressing the representation of the hash.
3. **Document References**: For each  $(b, z)$  pair, store a list of document IDs (or references) that contain the token corresponding to this hash.

Inverted Index Structure:

python

Copy

```
inverted_index = {
    (b1, z1): [doc_id1, doc_id2, ...],
    (b2, z2): [doc_id3, doc_id4, ...],
    ...
}
```

Steps:

1. **Tokenization and Hashing**: For each document, tokenize the text and convert each token into a 64-bit hash.
2. **Compression**: Convert the 64-bit hash into  $(b, z)$  where:
  - $b$  is the integer value of the first  $k$  bits of the hash.
  - $z$  is the number of trailing zeros in the hash.
3. **Update Inverted Index**: For each  $(b, z)$  pair, append the document ID to the corresponding list in the inverted index.

Example:

- Suppose we have a token with a 64-bit hash: `0b110101...0000` (with 4 trailing zeros).

- If  $k = 8$ , then  $b$  is the integer value of the first 8 bits (e.g., `0b11010101`).
- $z$  is 4 (number of trailing zeros).
- The inverted index entry would be:  $(b, 4): [\text{doc\_id}]$ .

#### Efficiency Considerations:

- **Space Efficiency:** By using  $(b, z)$  instead of the full 64-bit hash, we reduce the storage overhead.
- **Lookup Efficiency:** The inverted index allows fast retrieval of documents containing a specific token.

## Task 2: Algorithm to Restore Document Tokens Using $(b, z)$ and Cardinality Constraint

Given the  $(b, z)$  pairs associated with a document and an estimate of the number of distinct tokens (cardinality), we can use **constraint programming** to reconstruct the original tokens. The idea is to generate all possible 64-bit hashes that match the  $(b, z)$  constraints and then use the cardinality to narrow down the correct set of tokens.

#### Steps:

##### 1. Generate Candidate Hashes:

- For each  $(b, z)$  pair, generate all possible 64-bit hashes that:
  - Have the first  $k$  bits equal to  $b$ .
  - Have exactly  $z$  trailing zeros.
- This can be done by fixing the first  $k$  bits and the last  $z$  bits, and then enumerating all possible combinations for the remaining bits.

##### 2. Apply Cardinality Constraint:

- The number of distinct tokens in the document is known (cardinality). This means that the number of unique  $(b, z)$  pairs should match the cardinality.
- Use constraint programming to ensure that the generated set of hashes satisfies this constraint.

##### 3. Reconstruct Tokens:

- Once the candidate hashes are generated and validated, map them back to the original tokens using the tokenizer (if possible) or use them directly as the token representations.

#### Example:

- Suppose we have a document with the following  $(b, z)$  pairs:  $[(b_1, z_1), (b_2, z_2), (b_3, z_3)]$  and a cardinality of 3.
- For each  $(b, z)$  pair, generate all possible 64-bit hashes that match the constraints.
- Ensure that the total number of unique hashes matches the cardinality (3 in this case).
- The final set of hashes represents the tokens in the document.

#### Constraint Programming Approach:

- Use a constraint solver (e.g., Google OR-Tools, Z3, or any other CP solver) to model the problem:
  - Variables: The bits of the 64-bit hash (excluding the fixed  $k$  leading bits and  $z$  trailing zeros).
  - Constraints:
    - The first  $k$  bits must match  $b$ .
    - The last  $z$  bits must be zeros.
    - The total number of unique hashes must match the cardinality.
- Solve the model to find the valid set of hashes.

## Example Code (Pseudocode):

python

Copy

```
from constraint import Problem
```

```
def restore_tokens(b_z_pairs, cardinality, k):
    problem = Problem()
```

```
    # Variables: bits of the hash (excluding fixed bits)
```

```
    for i, (b, z) in enumerate(b_z_pairs):
```

```
        problem.addVariable(f"hash_{i}", generate_candidate_hashes(b, z, k))
```

```
    # Constraint: number of unique hashes must match cardinality
```

```
    problem.addConstraint(lambda *hashes: len(set(hashes)) == cardinality, [f"hash_{i}" for i in range(len(b_z_pairs))])
```

```
    # Solve the problem
```

```
    solutions = problem.getSolutions()
```

```
    # Return the restored tokens (hashes)
```

```
    return solutions
```

```
def generate_candidate_hashes(b, z, k):
```

```
    # Generate all possible 64-bit hashes with first k bits = b and last z bits = 0
```

```
    # This is a placeholder for the actual implementation
```

```
    return [hash for hash in possible_hashes if (hash >> (64 - k)) == b and (hash & ((1 << z) - 1)) == 0]
```

## Efficiency Considerations:

- **Scalability:** The number of candidate hashes grows exponentially with the number of free bits ( $64 - k - z$ ). This approach may not scale well for large values of  $k$  and  $z$ .
- **Optimization:** Use pruning techniques or heuristics to reduce the search space.

## Summary:

1. **Inverted Index:** Use  $(b, z)$  as keys in an inverted index to efficiently store and retrieve document references.
2. **Token Restoration:** Use constraint programming to generate and validate candidate hashes based on  $(b, z)$  pairs and cardinality constraints.

This approach balances storage efficiency with the ability to reconstruct document tokens when needed.

can we improve this solution by adding token (hash) frequency in all documents. in this case the tokens with lower frequencies would be more relevant to the specific document and we can use this fact as criteria for the problem solution in constraint model.



Yes, incorporating **token frequency** across all documents can significantly improve the solution. Tokens with lower frequencies are more specific to certain documents and can act as strong discriminators during the token restoration process. By using token frequency as an additional criterion, we can prioritize rare tokens in the constraint model, making the solution more accurate and efficient.

## Improved Solution: Incorporating Token Frequency

Key Idea:

- **Token Frequency:** For each token hash, maintain its frequency across all documents (i.e., how many documents contain this token).
- **Relevance:** Tokens with lower frequencies are more relevant to specific documents and can be used to narrow down the solution space in the constraint model.

## Updated Inverted Index Structure

The inverted index will now store the frequency of each token hash in addition to the document references:

python

Copy

```
inverted_index = {
    (b1, z1): {"frequency": freq1, "documents": [doc_id1, doc_id2, ...]},
    (b2, z2): {"frequency": freq2, "documents": [doc_id3, doc_id4, ...]},
    ...
}
```

Steps to Build the Inverted Index:

1. **Tokenization and Hashing:** Tokenize each document and convert tokens into 64-bit hashes.
2. **Compression:** Convert each hash into  $(b, z)$  as before.
3. **Update Inverted Index:**
  - For each  $(b, z)$  pair, update the frequency (increment by 1 for each document containing the token).
  - Append the document ID to the list of documents for the  $(b, z)$  pair.

## Constraint Model with Token Frequency

Key Changes:

1. **Frequency-Based Prioritization:**
  - Tokens with lower frequencies are more likely to be unique to the document and should be prioritized in the constraint model.
  - Use frequency as a weight or cost function to guide the search for valid tokens.
2. **Objective Function:**
  - Minimize the total frequency of the selected tokens. This ensures that the solution favors rare tokens, which are more discriminative.
3. **Constraints:**
  - The number of unique tokens must match the document's cardinality.
  - Each  $(b, z)$  pair must correspond to a valid token hash.

## Algorithm Steps:

1. **Input:**
  - A list of  $(b, z)$  pairs associated with the document.

- The cardinality of the document (number of distinct tokens).
  - Token frequencies from the inverted index.
- 2. Generate Candidate Hashes:**
- For each  $(b, z)$  pair, generate all possible 64-bit hashes that match the constraints (first  $k$  bits =  $b$ , last  $z$  bits = 0).
- 3. Assign Frequencies:**
- For each candidate hash, look up its frequency in the inverted index. If the hash is not in the index, assume a high frequency (e.g., infinity) to discourage its selection.
- 4. Constraint Programming Model:**
- Variables: The candidate hashes for each  $(b, z)$  pair.
  - Constraints:
    - The number of unique hashes must match the cardinality.
  - Objective: Minimize the total frequency of the selected hashes.
- 5. Solve the Model:**
- Use a constraint solver to find the set of hashes that satisfies the constraints and minimizes the total frequency.

## Example Code (Pseudocode):

python

Copy

```

from constraint import Problem, MinConflictsSolver

def restore_tokens_with_frequency(b_z_pairs, cardinality, k, inverted_index):
    problem = Problem(MinConflictsSolver())

    # Generate candidate hashes for each (b, z) pair
    candidates = []
    for b, z in b_z_pairs:
        hashes = generate_candidate_hashes(b, z, k)
        candidates.append(hashes)

    # Add variables and their frequencies
    for i, hashes in enumerate(candidates):
        problem.addVariable(f"hash_{i}", hashes)

    # Constraint: number of unique hashes must match cardinality
    problem.addConstraint(lambda *hashes: len(set(hashes)) == cardinality, [f"hash_{i}" for i in range(len(b_z_pairs))])

    # Objective: minimize total frequency of selected hashes
    def frequency_cost(*hashes):
        total_freq = 0
        for hash_val in hashes:
            # Look up frequency in inverted index
            b_z = (hash_val >> (64 - k), count_trailing_zeros(hash_val))
            if b_z in inverted_index:
                total_freq += inverted_index[b_z]["frequency"]
            else:
                total_freq += float('inf') # Penalize unknown hashes
        return total_freq

```

```

problem.addObjective(frequency_cost, [f"hash_{i}" for i in range(len(b_z_pairs))])

# Solve the problem
solution = problem.getSolution()

# Return the restored tokens (hashes)
return [solution[f"hash_{i}"] for i in range(len(b_z_pairs))]

def generate_candidate_hashes(b, z, k):
    # Generate all possible 64-bit hashes with first k bits = b and last z bits = 0
    # This is a placeholder for the actual implementation
    return [hash for hash in possible_hashes if (hash >> (64 - k)) == b and (hash & ((1 << z) - 1)) == 0]

def count_trailing_zeros(hash_val):
    # Count the number of trailing zeros in the hash
    return (hash_val & -hash_val).bit_length() - 1

```

## Benefits of Using Token Frequency:

### 1. Improved Accuracy:

- Rare tokens are more likely to be unique to the document, reducing ambiguity in token restoration.

### 2. Efficiency:

- The constraint model converges faster by prioritizing low-frequency tokens.

### 3. Robustness:

- High-frequency tokens (e.g., stopwords) are less likely to be selected, reducing noise in the solution.

## Example Walkthrough:

### • Input:

- (b, z) pairs: [(b1, z1), (b2, z2), (b3, z3)]
- Cardinality: 3
- Inverted index with frequencies:
  - (b1, z1) : frequency = 2
  - (b2, z2) : frequency = 1
  - (b3, z3) : frequency = 5

### • Candidate Hashes:

- For each (b, z), generate all possible hashes.

### • Constraint Model:

- Select 3 unique hashes with the minimum total frequency.

### • Solution:

- The model selects the hashes with frequencies 1, 2, and 5 (total = 8), favoring the lowest frequencies first.

## Conclusion:

By incorporating token frequency into the constraint model, we can significantly improve the accuracy and efficiency of token restoration. This approach leverages the discriminative power of rare tokens, making it a robust solution for the problem.

some small correction to the provided solution.

$(b, z)$  is one-to-many map. we can calculate frequency only for tokens and tokens with the same  $(b, z)$  can have different frequencies. we can use this fact to narrow scope of selected tokens that may lower dimensionality of the problem by picking only tokens with lower frequencies



You're absolutely correct! The  $(b, z)$  representation is a **one-to-many map**, meaning multiple tokens (hashes) can share the same  $(b, z)$  pair but have different frequencies. By leveraging this fact, we can **narrow the scope of candidate tokens** by prioritizing those with lower frequencies. This reduces the dimensionality of the problem and improves the efficiency and accuracy of the constraint model.

## Updated Approach: Leveraging Token Frequencies within $(b, z)$ Groups

Key Idea:

- For each  $(b, z)$  pair, there may be multiple tokens (hashes) with different frequencies.
- Tokens with lower frequencies are more discriminative and should be prioritized in the constraint model.
- Instead of considering all possible hashes for a  $(b, z)$  pair, we can **pre-filter** the candidate hashes to include only those with the lowest frequencies.

## Steps to Improve the Solution:

### 1. Pre-Filter Candidate Hashes:

- For each  $(b, z)$  pair, retrieve all tokens (hashes) that map to it.
- Sort these tokens by frequency (ascending order).
- Select the top  $N$  tokens with the lowest frequencies for further consideration. This reduces the search space.

### 2. Update Constraint Model:

- Use the pre-filtered candidate hashes in the constraint model.
- The objective remains to minimize the total frequency of the selected tokens.

### 3. Cardinality Constraint:

- Ensure that the number of unique tokens selected matches the document's cardinality.

## Updated Algorithm:

Input:

- A list of  $(b, z)$  pairs associated with the document.
- The cardinality of the document (number of distinct tokens).
- A mapping from  $(b, z)$  to tokens (hashes) and their frequencies.

Output:

- A set of tokens (hashes) that best match the document, given the constraints.

## Pseudocode:

```
python

from constraint import Problem, MinConflictsSolver

def restore_tokens_with_frequency(b_z_pairs, cardinality, k, bz_to_tokens):
    problem = Problem(MinConflictsSolver())

    # Pre-filter candidate hashes for each (b, z) pair
    candidates = []
    for b, z in b_z_pairs:
        # Get all tokens for this (b, z) pair
        tokens = bz_to_tokens.get((b, z), [])
        # Sort tokens by frequency (ascending) and select top N
        tokens_sorted = sorted(tokens, key=lambda x: x["frequency"])[::-10] # Adjust N as needed
        candidates.append([token["hash"] for token in tokens_sorted])

    # Add variables to the problem
    for i, hashes in enumerate(candidates):
        problem.addVariable(f"hash_{i}", hashes)

    # Constraint: number of unique hashes must match cardinality
    problem.addConstraint(lambda *hashes: len(set(hashes)) == cardinality, [f"hash_{i}" for i in range(len(b_z_pairs))])

    # Objective: minimize total frequency of selected hashes
    def frequency_cost(*hashes):
        total_freq = 0
        for hash_val in hashes:
            # Look up frequency in the pre-filtered candidates
            for b, z in b_z_pairs:
                tokens = bz_to_tokens.get((b, z), [])
                for token in tokens:
                    if token["hash"] == hash_val:
                        total_freq += token["frequency"]
                        break
        return total_freq

    problem.addObjective(frequency_cost, [f"hash_{i}" for i in range(len(b_z_pairs))])

    # Solve the problem
    solution = problem.getSolution()

    # Return the restored tokens (hashes)
    return [solution[f"hash_{i}"] for i in range(len(b_z_pairs))]

# Example data structure for bz_to_tokens
bz_to_tokens = {
    (b1, z1): [{"hash": hash1, "frequency": freq1}, {"hash": hash2, "frequency": freq2}, ...],
    (b2, z2): [{"hash": hash3, "frequency": freq3}, {"hash": hash4, "frequency": freq4}, ...],
    ...
}
```

## Explanation of Changes:

### 1. Pre-Filtering Candidates:



- For each  $(b, z)$  pair, we retrieve all tokens and sort them by frequency.
  - We select the top  $N$  tokens with the lowest frequencies (e.g.,  $N = 10$ ). This reduces the search space while preserving the most discriminative tokens.
- 2. Frequency-Based Objective:**
- The objective function minimizes the total frequency of the selected tokens. This ensures that the solution favors rare tokens.
- 3. Efficiency:**
- By pre-filtering candidates, the constraint model operates on a smaller set of variables, making it more efficient.

## Example Walkthrough:

Input:

- $(b, z)$  pairs:  $[(b1, z1), (b2, z2), (b3, z3)]$
- Cardinality: 3
- $bz\_to\_tokens$  :
  - $(b1, z1)$  :  $[\{"hash": hash1, "frequency": 2\}, \{"hash": hash2, "frequency": 5\}]$
  - $(b2, z2)$  :  $[\{"hash": hash3, "frequency": 1\}, \{"hash": hash4, "frequency": 3\}]$
  - $(b3, z3)$  :  $[\{"hash": hash5, "frequency": 4\}, \{"hash": hash6, "frequency": 6\}]$

Pre-Filtering:

- For each  $(b, z)$ , select the top  $N = 2$  tokens with the lowest frequencies:
  - $(b1, z1)$  :  $[hash1 (freq=2), hash2 (freq=5)]$
  - $(b2, z2)$  :  $[hash3 (freq=1), hash4 (freq=3)]$
  - $(b3, z3)$  :  $[hash5 (freq=4), hash6 (freq=6)]$

Constraint Model:

- Select 3 unique hashes with the minimum total frequency.
- Possible solution:  $[hash1 (freq=2), hash3 (freq=1), hash5 (freq=4)]$  (total frequency = 7).

## Benefits of the Updated Solution:

- 1. Reduced Dimensionality:**
  - By pre-filtering tokens, the search space is smaller, making the constraint model more efficient.
- 2. Improved Accuracy:**
  - Prioritizing low-frequency tokens ensures that the solution is more discriminative and relevant to the document.
- 3. Scalability:**
  - The approach scales better for large datasets, as the number of candidate tokens is controlled.

This updated solution effectively leverages the one-to-many relationship between  $(b, z)$  and tokens, as well as token frequencies, to improve the token restoration process.