# Assignment-1

## **Problem Statement**:

Given a data set which has transactions of Item sets. The task is to do lossless compression of data using frequent items sets derived from Algorithms like Apriori and FP Growth.

## **Our Approach**:

We are using the FP Growth algorithm in which we will prepare a Frequent Pattern Tree and mine the frequent patterns above a static threshold AKA minimum support. After getting the frequent patterns, we are comparing it with each transaction and replacing the frequent pattern that can contribute to better compression. A compression parameter is calculated as

**{Size of item sets * support * No of transactions}**

Since No of transactions is constant for all item sets, it can be removed.

**compression parameter = {Size of item sets * support}** is used to order Item sets. Each item set is encoded, we are using base16 (hex) encoding with Elias gamma encoding. Ellias gamma to reduce clashes and base 16 to condense the binary digits. In a version, we used base84 encoding to accommodate many item sets.

Why 84? 84 characters are visible in ASCII encoding. We removed characters like space and built base 84 number system.

Usage of the above encoding depends upon tradeoff between time and compression. If you want more compression, go for base84, otherwise for the time being, go for elias gamma and base16 encoding.

## **Understanding the code:**

The code checks command-line arguments to determine whether to perform compression or decompression based on the input flag ('C' for compression, 'D' for decompression).

If compression is selected, the code calls **compress_main** and prints compression-related statistics.

If decompression is selected, the code calls **decompress_main**.

The code checks command-line arguments to determine whether to perform compression or decompression based on the input flag ('C' for compression, 'D' for decompression).
If compression is selected, the code calls **compress_main** and prints compression-related statistics.
If decompression is selected, the code calls **decompress_main**.

The "**compress"** function applies the FP-Growth algorithm to find frequent patterns, compresses transactions using pattern indexing, and saves the compressed data along with the mapping information. The compression process is optimized through parallel processing and efficient storage of frequent pattern-to-index mappings.

**Reading and Preprocessing**:

- Read the input dataset from the specified **inputPath**.
- For each line (transaction) in the dataset, split it into items and store them in the "**transactions"** list.
- Calculate the initial size of the data before compression.

**FP-Growth and Frequent Pattern Mining**:

- Determine the number of transactions in the dataset (**nr**).
- Calculate a suitable batch size based on the number of transactions.
- Process transactions in batches to avoid memory issues.
- Transform the transactions into a binary matrix using **TransactionEncoder**.
- Construct the FP-Tree and mine frequent itemsets using FP-Growth.
- Filter out frequent patterns with lengths less than 2.

**Indexing and Mapping Frequent Patterns**:

- Calculate the frequency of frequent patterns based on support.
- Create a mapping (**pattern_to_index**) between frequent patterns and their corresponding indices.
- Sort the mapping by pattern length to optimize subsequent processing.

**Parallel Compression**:

- Determine the number of CPU cores available for parallel processing.
- Define a partial function **partial_process** that takes a transaction set and the sorted pattern-to-index mapping. This function will be used for parallel processing.
- Create a multiprocessing pool and use **map** to apply the **partial_process** function to each batch of transactions. The result is a list of compressed transaction sets (**compressed_lst**).

**Saving Pattern-to-Index Mapping**:

- Depending on the number of transactions, either save the sorted pattern-to-index mapping to a single file or to multiple batch files for large datasets.
- Calculate the size of the data after compression (**end_size**).
- Calculate and print the compression ratio as a percentage.
- Write the compressed transaction sets to the specified **outputPath**.

The **decompress** function uses pattern-to-index mapping to decompress the compressed dataset. It reads the compressed data, maps encoded indices back to frequent pattern sets, and writes the reconstructed original transactions to an output file. This process effectively restores the original transactional dataset from the compressed form.

**Loading Pattern-to-Index Mapping**:

- Load the pattern-to-index mapping (**sorted_pattern_to_index**) from the **sorted_pattern_to_index.pkl** file that was saved during the compression process. This mapping is crucial for reversing the compression.

**Reading Compressed Data**:

- Open the compressed input file (**inputPath**) and read each line, which represents a compressed transaction.

**Reconstructing Original Transactions**:

- Create an empty list called **decomp_lst** to store the reconstructed original transactions.

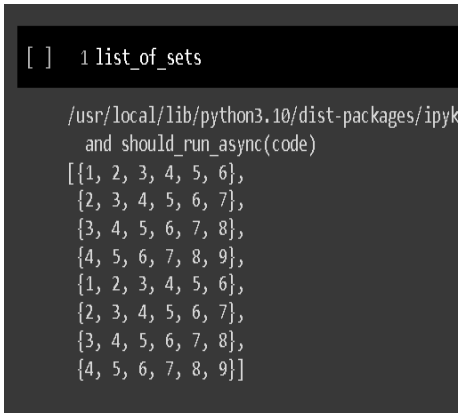
- Iterate through each compressed transaction (**comp_set**) in the list of compressed sets.
- For each item in the compressed set, check if it's a special value ('nan') or a hex-encoded frequent pattern index.
- If the item is an index, use the reverse index (**rev_indx**) to map it back to the corresponding frequent pattern set, and add the items from that set to the **decomp_set**.
- If the item is not a special value or an index, add it directly to the **decomp_set**.
- After processing all items in the compressed set, add the **decomp_set** to the list of reconstructed transactions (**decomp_lst**).

**Writing Decompressed Data**:

- Use **pd.DataFrame(decomp_lst)** to create a DataFrame from the list of reconstructed transactions.
- Write the DataFrame to the specified **outputPath** as a tab-separated text file.
- This file now contains the reconstructed original transactions, effectively decompressing the data.

## Proof Of Concept:

We took a file of 8 transactions as shown below.



Convert into one hot encoding in the dataframe .

```
[ ]  1 dfr.values

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel
  and should_run_async(code)
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True, False,  True, ...,  True,  True,  True],
       [ True, False,  True, ...,  True,  True,  True],
       ...,
       [False,  True,  True, ..., False,  True,  True],
       [False,  True,  True, ..., False,  True,  True],
       [False,  True,  True, ..., False,  True,  True]])
```

Use FPGrowth and Hex encoding to convert frequent items sets into encodings .

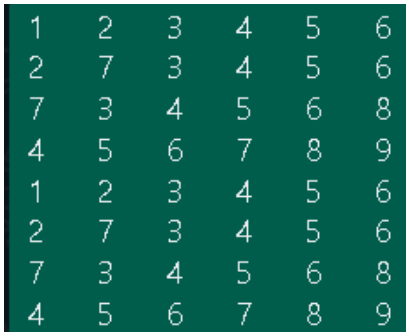
```
1 sorted_pattern_to_index

/usr/local/lib/python3.10/dist-packa
  and should_run_async(code)
{frozenset({3, 4, 5, 6}): '0x4',
 frozenset({4, 5, 6, 7}): '0x8',
 frozenset({4, 5, 6}): '0x0',
 frozenset({3, 4, 5}): '0x2',
 frozenset({3, 4, 6}): '0x2',
 frozenset({3, 5, 6}): '0x3',
 frozenset({4, 5, 7}): '0x5',
 frozenset({4, 6, 7}): '0x6',
 frozenset({5, 6, 7}): '0x7'}
```

Compressed file is as such.

ass12.py | C_small(2).dat

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 0x4 |
| 2 | 2 | 7 | 0x4 |
| 3 | 8 | 7 | 0x4 |
| 4 | 8 | 9 | 0x8 |
| 5 | 1 | 2 | 0x4 |
| 6 | 2 | 7 | 0x4 |
| 7 | 8 | 7 | 0x4 |
| 8 | 8 | 9 | 0x8 |

Recovering compressed dataset




Order of transactions is maintained but order of itemset is lost, which is not concerning.

**Team Contribution**

The team members have contributed to the assignment as mentioned below.

M. Yagnesh (2023AIB2069) : 30
Potluri Krishna Priyatham (2023AIB2084) : 35
Abhishek Goyal (2023AIB2073) : 35