Assumptions

* I optimize for speed not storage (e.g. using hash and ArrayList to get constant access time)
* Words are separated by one or more white spaces
* A sentence may appear more than once in the groups (sentence matches more than one pattern)

Data Structure

* The main data structure is a hash table for storing all the patterns and their appearances. In each entry the key is the pattern string and the value is a list of the appearances
* The pattern string is a sentence with a wildcard that can be replaced by an arbitrary word, the wildcard is marked as "**%s**" e.g.

***%s is eating at a diner***

* Appearance is a pair of timestamp and the word that replaced the wildcard

Flow: reading input file

1. The input file is read line by line
2. The timestamp is parsed and is saved as a Date object
3. The rest of the line is split to words
4. From each record all the possible pattern strings are generated (by replacing in turn each word in the sentence with a wildcard)
5. Pattern strings that are not already exist in the patterns hash table are added
6. The timestamp and the word that replaces the wildcard are added to the appearance list

Flow: generating output file

1. Iterating over the patterns hash table
2. For every pattern that has more than 1 appearances, we reconstruct and write to the output file the original record (i.e. concatenating the date from the appearance object as a string, followed by the pattern string where the wildcard is replaced with the word stored in the appearance object)
3. In addition we store the appearances' words in an array
4. After iterating over the appearances, we write a new line with the words of the array

Complexity

* For each line we iterate over the words of the sentence. If we consider that the number of the words in the sentences is capped by a constant (reasonable assumption) then the complexity is O(N) where N is the number of lines. Otherwise it is O(N \*M) where M is the number of words in the sentence

Scale

* The algorithm can scale quite a lot as is. For long input files the patterns can be saved in 2 database tables one for patterns and the other for appearances

Improvements

* In terms of performance one improvement that can be skipping words that are "not interesting" like: ***and at an*** etc.
* Another reduction in storage can come from compressing the pattern strings. Since many of the pattern strings are similar compressions algorithms like Huffman tree may reduce significantly the storage allocated for storing strings.
* We can make the analysis more robust by overlooking spelling mistakes
* If the goal is to find in the input file more patterns and more sophisticated patterns then we can
  + Support multi-word tokens (e.g. "M goes to the bank" and "M goes to the post office" will share the same pattern)
  + Identify expressions like time or location expression (street addresses)
  + Patterns based on meaning (e.g. "M has lunch at home" and "M has lunch at noon" are not the same pattern because one is a time expression and the other is a location expression)
  + Identify activity sequences and daily routines (e.g. M runs every day between 7AM and 9AM)
  + Identify relationships between tokens and real life objects (e.g. deduct that the token "vehicle 87-765-98" and "red ford mustang" refer to the same object)