Kacy Lombard
Analysis and Tradeoffs

Database Design:

I would choose to implement a single relational, SQL database table with a schema that looks like this: [id, employer, mask, firstName, lastName, amountExpected, username]. (SQL was an arbitrary decision here to go in a specific direction; however, it is worth noting that a NoSQL implementation would be more horizontally scalable (no need for database partitioning when scaling) and potentially more cost-efficient if the user base were to increase significantly. Although, a dynamic schema is most-likely not necessary for this specific table so SQL works well. Furthermore, SQL is ACID compliant which is necessary for processing payments and storing important user data. The primary key in this scenario would be a composite key that looks like PRIMARY KEY(employer, firstName, lastName). Additional indices would most likely not be necessary here, especially with the following tokenization method.

Another way this schema could be organized (which is most likely more efficient and would maintain user privacy), is to tokenize a user's employer, firstName, lastName, and mask into a single, unique ID that ensures both privacy and confirms a 100% identical match, with only a single query. It also eliminates the need for a "composite" primary key to identify uniqueness, as the token can be constructed from each entry in the payments.csv, and then used to query that exact user's record, while keeping the user's data (firstName, lastName, employer, mask and potentially more fields in the future such as SSN completely private and tokenized). This schema would look something like this: [id, token, amountExpected, username]. In my solution, I essentially did something similar by doing a basic string concatenating tokenization, where each token was "employer" + "firstName" + "lastName", for confidently identifying unique matches. The primary key in this schema would be the token.

```
CREATE DATABASE loans;
USE loans;
CREATE TABLE records (id INT NOT NULL AUTO_INCREMENT, token varChar(255)
UNIQUE NOT NULL primaryKey(token), amountExpected DECIMAL(8,2), username
UNIQUE NOT NULL varChar(64));
```

I would also have a separate payment table, storing all past payments for liability and legal purposes. The schema for this would look something like: [id, token, amount, timestamp]

In order to ensure the services are scalable, and to not incur future downtime in order to handle scaling, I would utilize consistent hashing partitioning criteria. The hash function I will use is the MD5 algorithm. I would partition my table into 10 separate parts each with their own server, with each part containing 1/10th of the different ids or tokens. This should allow for faster lookup and record matching in our overall table (as table partitions are smaller than the whole).

In order to efficiently interact with the database, I would modify my payment processing function to tokenize each entry based upon the aforementioned fields, and then simply query the correct database partition, based on which of the 10 partitions the token falls into. Upon retrieving the matching record, I would simply compute the remaining amountExpected and update the respective entry in the table. I would also then add this new payment entry into my payments table. The time complexity of this algorithm is simply O(n log(n)), if we assume database queries using the primary key are O(log(n)) and indexes are implemented with B-trees. This is because we have to iterate over payments.csv, and for each payment record, retrieve the user record, update it, and then store the new payment record in our payments table. Insert, update, and select should all be O(log(n)), but could vary based upon the specific database used.

My actual coded solution in Python 3 is O(n) because there is only one loop iterating over the payments, without any nested loops, and everything else utilizes lookup in dictionaries, which is an O(1) operation. Additionally, Python uses pointers for file I/O, so it is also O(1).