Distributed Data Management
Akka Actor Programming Hands-On

Task 3 – Password Cracking

Group: Robert'); DROP TABLE Students;–
(Joan Bruguera & Tom Braun)

Problem Statement

We are given a CSV file, containing multiple records. Each record corresponds to a user in an hypothetical system, and contains a SHA256 hash of the password of said user, along with the character set and password length used in said password. The objective is to crack (obtain the plaintext) of the password of each user.

In this hypothetical problem, each record also contains multiple hints in the form of SHA256 hashes. Each of those hint hashes is generated by hashing all except one of the characters in the password character set, and this missing character in the hint reveals that it is also not one of the characters in the password.

Our overall strategy

Our overall strategy:

- Cracking is done in batches
  - Only one batch is loaded/processed at a given moment

Our overall strategy

Our overall strategy:

- Cracking is done in batches
  - Only one batch is loaded/processed at a given moment
- **All hints for all records** in the batch are cracked

Our overall strategy

Our overall strategy:

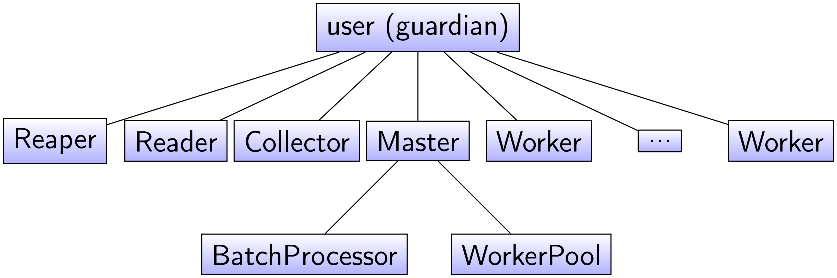
- Cracking is done in batches
  - Only one batch is loaded/processed at a given moment
- **All hints for all records** in the batch are cracked
- After all hints *for a given record* are cracked, the corresponding password is cracked (using the information in the hints)

Our overall strategy

Our overall strategy:

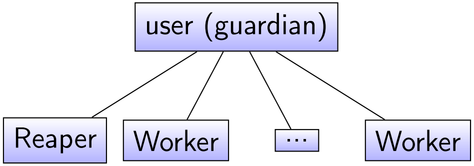
- Cracking is done in batches
  - Only one batch is loaded/processed at a given moment
- **All hints for all records** in the batch are cracked
- After all hints *for a given record* are cracked, the corresponding password is cracked (using the information in the hints)
- When all passwords in the batch are cracked, the system proceeds to the next batch

Master actor system



- **Reaper**, **Reader** & **Collector** mostly unchanged
- **Master**: Delegates worker registration messages to the worker pool and batches to the batch processor actors.
- **BatchProcessor**: Splits a batch into multiple work items for the worker actors and aggregates back the results
- **WorkerPool**: Routes and load balances the work items to the worker actors

Worker actor system



- **Workers** have two possible tasks:
  1. Crack a set of hint hashes:
     - **Input:** List of hint SHA256 hashes, possible characters, prefix
     - **Output:** Hashtable of (hint SHA256 hash $\rightarrow$ missing character) for the cracked hints
  2. Crack a password:
     - **Input:** Password SHA256 hash, possible characters, length
     - **Output:** Password SHA256 hash, password plaintext

Hint cracking

Since each hint consists of all `PasswordChars` except one, it is fundamentally a permutation problem. Given the hash of a hint, we can crack it as follows:

1. Iterate over all permutations of `PasswordChars`.
2. For each permutation, compute the SHA256 hash of all characters in the permutation except the last one.
3. If the hash matches, we have cracked the hint!

Hint cracking

Since each hint consists of all `PasswordChars` except one, it is fundamentally a permutation problem. Given the hash of a hint, we can crack it as follows:

1. Iterate over all permutations of `PasswordChars`.
2. For each permutation, compute the SHA256 hash of all characters in the permutation except the last one.
3. If the hash matches, we have cracked the hint!

E.g. in pseudocode:

```
def crack_hint(password_chars, hint_hash):
    for p in permutations(password_chars):
        candidate_hash = sha256(p[0:len(p)-2])
        if candidate_hash == hint_hash:
            # Return the missing character in the hint
            return p[len(p)-1]
    return None # Failed to crack hint
```

Hint cracking (cont.)

However, since **all records in the CSV file** have the same `PasswordChars`, we do *a lot* of redundant work if we apply the above algorithm for each single hint.

Instead, we can **simultaneously** crack many hint hashes in one iteration, by building a hashtable of hint hashes and checking whether the hash is in the hashtable.

Hint cracking (cont.)

However, since **all records in the CSV file** have the same `PasswordChars`, we do *a lot* of redundant work if we apply the above algorithm for each single hint.

Instead, we can **simultaneously** crack many hint hashes in one iteration, by building a hashtable of hint hashes and checking whether the hash is in the hashtable.

E.g. in pseudocode:

```
def crack_hints(password_chars, records):
    hint_hashes = set(h for h in r.hint_hashes for r in records)
    cracked_hint_hashes = dict()

    for p in permutations(password_chars):
        candidate_hash = sha256(p[0:len(p)-2])
        if candidate_hash in hint_hashes:
            # Store the missing character in the hint
            cracked_hint_hashes[candidate_hash] = p[len(p)-1]

    return cracked_hint_hashes
```

Distributing hint cracking

In order to efficiently parallelize and distribute the hint cracking, we need to figure a way to execute the process with many workers. This is done as follows:

1. The **BatchProcessor** partitions the hint cracking process evenly, using two-character prefixes of `PasswordChars`.
   - E.g. if `PasswordChars` is ABCDEFGHIJK, the prefixes are AB, AC, AD, ..., JH, JI, JK.
   - The first work item will iterate over the permutations of the form AB***********, the second on AC***********, etc..
   - $\rightarrow$110 work items are created.

Distributing hint cracking

In order to efficiently parallelize and distribute the hint cracking, we need to figure a way to execute the process with many workers. This is done as follows:

1. The **BatchProcessor** partitions the hint cracking process evenly, using two-character prefixes of `PasswordChars`.
   - E.g. if `PasswordChars` is ABCDEFGHIJK, the prefixes are AB, AC, AD, ..., JH, JI, JK.
   - The first work item will iterate over the permutations of the form AB***********, the second on AC***********, etc..
   - →110 work items are created.
2. Work items are routed and load balanced among the worker actors by the **WorkerPool**.
3. The **Workers** execute the hint cracking algorithm over their partition of the process and return the cracked hints to the **BatchProcessor**
4. The **BatchProcessor** receives back and aggregates the results.

Password cracking

Once all hints are cracked, cracking the passwords is straightforward as a combination problem. Given the hash of a password, we can crack it as follows:

1. Iterate over all combinations of the remaining `PasswordChars` after solving for the hints.
2. For each combination, compute the SHA256 hash of all characters in the combination.
3. If the hash matches, we have cracked the password!

Password cracking

Once all hints are cracked, cracking the passwords is straightforward as a combination problem. Given the hash of a password, we can crack it as follows:

1. Iterate over all combinations of the remaining `PasswordChars` after solving for the hints.
2. For each combination, compute the SHA256 hash of all characters in the combination.
3. If the hash matches, we have cracked the password!

E.g. in pseudocode:

```
def crack_password(remaining_password_chars, password_hash, password_length):
    for c in combinations(remaining_password_chars, password_length):
        candidate_hash = sha256(c)
        if candidate_hash == password_hash:
            # Return the password plaintext
            return c
    return None # Failed to crack password
```

Distributing password cracking

Distributing the password cracking is straightforward: One work item is created for every password to crack.

1. Once the **BatchProcessor** has received all cracked hints for a record in the batch, it generates a new work item to crack the password.
2. Work items are routed and load balanced among the worker actors by the **WorkerPool**.
3. The **Workers** execute the password cracking algorithm and return the cracked password to the **BatchProcessor**
4. The **BatchProcessor** gives the password plaintext to the **Collector** to be printed later.
5. Once all passwords are cracked, the **BatchProcessor** sends a message to the **Master**, which causes a new batch to be read.

Optimization

While we are doing parallel and distributed computation, **implementing the algorithms efficiently** is still very important for overall performance!

- Efficient permutation iteration is not trivial, but there are algorithms for it:
    - "Countdown QuickPerm Algorithm" (by Phillip Paul Fuchs)

Optimization

While we are doing parallel and distributed computation, **implementing the algorithms efficiently** is still very important for overall performance!

- Efficient permutation iteration is not trivial, but there are algorithms for it:
    - "Countdown QuickPerm Algorithm" (by Phillip Paul Fuchs)
- Avoid common Java performance pitfalls. E.g.:
    - Avoid concatatenating `Strings` (causes a copy)
    - Avoid repeatedly calling `String.getBytes()` (causes a copy)
    - `MessageDigest` instances can be reused
    - ...

Results

- **Good performance:** Even when running on a single node (typical laptop), the full process is very fast (e.g. <20 seconds with 4 workers).
- **Good scalability:** The master does few work, and the workload is split on many work items, so that it can be efficiently distributed over a moderately-sized cluster.

Weak points

1. We are not resistant to message loss or workers disconnections.
    - **Solution:** Add disconnection and retry logic.
2. We are not resistant to invalid / uncrackable inputs.
    - **Solution:** Add more validation logic.
3. The single `BatchProcessor` on the master system is relatively light, but it could become a bottleneck if the data set and the cluster becomes large.
    - **Possible solution:** Process multiple batches simultaneously, with multiple `BatchProcessors`.

Weak points

1. We are not resistant to message loss or workers disconnections.
   - **Solution:** Add disconnection and retry logic.
2. We are not resistant to invalid / uncrackable inputs.
   - **Solution:** Add more validation logic.
3. The single `BatchProcessor` on the master system is relatively light, but it could become a bottleneck if the data set and the cluster becomes large.
   - **Possible solution:** Process multiple batches simultaneously, with multiple `BatchProcessors`.

None of those problems are fundamental, and can be solved at the expense of somewhat increased effort and code complexity.