Thoughts on some simplifying assumptions you made and why.

- When scraping the name of actors and cast I only use last name, first name, first name + last name for the actors and did not combine the name suffix with the last name.
  - Issues that could arise:
    - For example for a name such as Robert Downey Jr. would return rober, downey, and roberdowney.
    - In order to search for Justin Randall Timberlake, the options are either justin, timberlake, or justintimberlake
  - Why?
    - Most of the time middle names and name suffix with last name are not used, so I thought this could only add unneeded complexity to the search system for now.

- I ran a script which prepossesses the data beforehand and stores it in a json, so when the server first runs it does not have to scrape the data from the first 1000 imbd movies.
  - Issues that could arise:
    - A new movie gets added to the IMBD top 1000/changes are made to a movie are not reflected in the search.
  - Why?
    - If the user wants to generate the changes they can by running scrape.py, and then rebuilding the docker. This choice was made to save time for the person attempting to test the code.

- Using lowercase only
  - Issues:
    - Rachael McAdams → rachaelmcadams, rachael, mcadams
  - Why?
    - Would lead to more simplicity and uniformity when using the search app for the time being.

Improve performance, scale, quality?

If building at scale we should use a distributed hash table sharding on the hash of each query term mod 256 and use an aggregate server to get the intersection of the results. For example let's say query=hanks&query=tom, we would hash('hanks) % 256 and hash('tom') % 256, get each result and return their intersection using the aggregate server.
We would make sure to have replicas of the DHT for reading/writing purposes as well if one dies.
We would save the data into a nosql/sql db as well based on our requirements.
Location table, cast table, etc.

Depending on the politeness policy we can add more threads each scraping a section of the total number of movies. For instance, thread1 can scrape data for movies from 0-250, thread2 can scrape data for movies 251-500, etc..

Another way we can do this is to have different threads work on different aspects of the scraping process. For instance one thread/process (or more than one thread/process depending on how you design it) can search through each movie and scrape the cast_url, location_url etc. and send each type of url (cast as one type and location as another, this can be scaled to many different url types) to different asynchronous queues. There will be worker threads/processes which will grab the urls from each of the queues. More specifically, one group of worker threads/processes would perform the scraping of cast_urls by grabbing cast_urls from the cast_url queue, other group of worker threads/processes would perform the scraping of location_urls by grabbing location_urls from the location_url queue. The number of threads/processes would depend on the politeness policy.

If we are repeating this process (running this every n minutes) looking for changes in the Top1000MBD movies we can store a checksum corresponding to each cast_url, location_url we search through and store in a DHT as well. When the cache_url/location_url worker first grabs the url from the queue they will get the html given by the url, calculate a checksum, and see if it matches anything in DHT. If it does, the work can grab the next message/url, otherwise it must scrape the page. Another option would be to use a bloom filter.

For both scenarios, searching and dedupe testing, it depends on how large the data that we are working with. If it is very large we might have to use a LRU based cache to store the more recently used data, while storing the rest in either a sql or no-sql db depending on the other requirements.