**How does login work?**

<insert login page waley page ka screenshot here>

<insert flow diagram for the below thing>

1. The passwords are stored after passing them through MD5 hash function.
2. After the user enters the username and password, the entered password is passed through MD5 hash function.
3. The table “users” is queried for the username and the hashed password.
4. If a match is found, session variables for userid and username are set and the user is redirected to the home page.
5. If no match is found, an error is shown for the same.
6. <insert error wala thing ka screenshot>

**How does registration work?**

<insert registration page waley page ka screenshot here>

<insert flow diagram for the below thing>

1. On the registeration page, the user enters his/her desired username and password. The password is to be typed twice to confirm and match.
2. When the user clicks the Sign Up button, the form is posted to the same page.
3. If the passwords don’t match, the session variable for message is set as “The two passwords do not match”. The password matching can be done using JavaScript dynamically too. That is a faster method than using php for the same.
4. If the length of the password is lesser than 8, the session variable for message is set as “The password length is less than 8”. We can also check for various parameters for the strength of the password like existence of at least one capital character, at least one special character and one number.
5. If the length of the username is lesser than 2, the session variable for message is set as “The Username is too small”.
6. If none of 3), 4) or 5) happen then a query is made to the “users” table to insert into it the username and MD5 hashed password. The userid is set as an Auto increment variable.
7. In the main body of the HTML page, we run another php snippet to check if the session variable for message is set or not. If it is set, then we show the error message accordingly.

<insert some images showing the error messages shown in red on the sign up page>

**How does search work?**

<insert an image of the search page>

<insert flow diagram for the same>

1. The user can type the name of the movie or part of it to search the movie database.
2. The search can also be done on the basis of the genre of the movie, for example: Comedy, Romance, etc.
3. The user enters his/her query into the search box and clicks on enter button or on the magnifying glass next to the seach box.
4. First of all, the length of the search query is checked and if it is too less then a JavaScript Alert pops up telling the user the same.
5. If the search query is greater than the required length and the user had selected search by name then a query is made to the table “moviedata” asking for all the movies whose name has the present search query as a substring. The search results are returned sorted in decreasing ordering of their average ratings and in case of a tie by the number of users who have rated.
6. If the search query is greater than the required length and the user had selected search by genre then a query is made to the table “moviedata” asking for all the movies whose genre is the same as that in the search query. The search results are returned sorted in decreasing ordering of their average ratings and in case of a tie by the number of users who have rated.
7. The filetered and sorted search results are printed via php as a HTML table.
8. If the user has already rated any of the returned movies, that movie name is shown in green otherwise it is shown in red.
9. The movie names are hyperlinks, clicking on which will lead the user to a page where he or she can add ratings for a movie.

<insert image of the js popup which appears when query length is too less>

**How does the rating system work?**

<insert image of the rating system with stars>

<insert flow diagram for the same>

1. When the user clicks on any movie from the search results, he or she is directed to a page where the user can modify (if already rated by the user) or enter his or her rating for that movie.
2. There are ten stars which change colour when hovered upon. The ratings work from the integers 1 to 10. Once the user clicks on a star and clicks on submit, his rating is entered. The stars can be thought of as radio buttons. If the user selects the xth star, it means the user wants to set the rating for this movie as x.
3. The user can change his or her rating multiple times by clicking on any star and only the final click will be counted for change.
4. The user rating is taken and a query is made to the “ratings” table to insert a new row.
5. The new row is inserted into the “ratings” table and then a query is made to the “moviedata” table to modify the average rating and the number of users rated.
6. Once the two values are modified in the “moviedata” table, the tab is killed and the user is now back to the search result page.
7. The user can now reload the search results and see that his or her latest rated movie is now shown in green colour if it was red before.

<insert an image with 10 stars rated>

**How does method 1 work?**

<insert a flow diagram for method 1>

1. This method is based on the genres of the movies that the user has rated.
2. The first query is made to the table “ratings” to find which movies has the user rated.
3. Once the movieids are known, queries are made to the table “moviedata” to find what genres these movies are.
4. The genre attribute for the table “moviedata” actually stores one single string with consecutive genres separated by a ‘|’.
5. We make an associative array in php where the key is the genre name and the value is the count of the occurrence of the genre name in all the movies rated by the users. The associative array can be seen a function which maps keys to values.
6. We decide upon how many top values we want to display for each genre. For our implementation, we decided to display top 3 matches for each genre.
7. We iterate over each genre type that the user has rated a movie for.
8. Now a query is made to the table “moviedata” for the movieid, moviename, average rating and genre selected from 6) of all the movies with more than a threshold number of ratings.
9. For each movie that 7) returns, we make a sorted insert into the three average rating values we decided upon. We keep track of the names of the movies too.
10. We print the results in sorted order for all the genres the user has rated for.
11. One thing to note here is that instead of iterating over all the genres for which the user has rated a movie, we can decide on a threshold value x. We iterate over only those genres for which the user has at least an average rating of x. This makes sense because the user might have negatively rated movies of a particular genre.
12. This method is surprisingly fast and works well since we are not looking at all the movies but only of those genres that are relevant to the users.

**How does method 2 work?**

<insert a flow diagram for method 2>

1. This method is based on user – user collaborative filtering. The basic idea behind this is that if two users have agreed on something in the past, then they will agree on things in the future too. For the current user, we find the most similar user. The most similar user is the user whose ratings are closest to our current user.
2. For this we define a similarity quotient. For our implementation, we have used the reciprocal of the Eucledian distance between the ratings vector of the two users for comparison.
3. For our current user, we store the movieids and the corresponding ratings in an two arrays. We also need a marked array to know which movies for our current user have been matched in any one iteration of the for loop.
4. We initialise max\_similarity and max\_similarity\_user as -1. After the full iteration, these will contain the maximum value of the similarity quotient and the userid of the user with which this similarity was achieved.
5. We loop over all the users and make a query to the table “ratings” to get the movieids and ratings for the current user.
6. We find the Eucledian distance between the rating arrays of the two users.
7. If a movie hasn’t been rated by a user, we take his or her rating for that movie to be 0.
8. After using the marked array to check for movies rated by the current user which didn’t find any match in this iteration, we update max\_similarity and max\_similarity\_user whenever needed.
9. In the end, we show the user those movies of the max\_similarity\_user which he or she hasn’t yet rated.

10) The time complexity of this method is O(n\*m\*t) where n is the number of users, m is the number of movies rated by our current user and t is the average number of movies rated by each user.

**How does method 3 work?**

<insert a flow diagram for method 3>

1) This method is an improvement over method 2 on the basis of implementation.

2) In method 2, we made two arrays for the current user: one to store movieids and the other to store the corresponding ratings.

3) In each iteration of the loop over all the users, we first found out which movies the user had rated and then searched for that movieid in the array of our current user.

4) If a match existed, we computed the distance between these two and squared it.

5) If a match didn’t exist, we let the rating of current user on this movie to be 0.

6) This led to O(m) extra computation in each iteration. Since we had to check if the movie existed in the movie array of the current user and decide accordingly.

7) Now what we can do is to use an associative array for the current user. The key is the movieid and the value is his or her rating on that movie.

8) In this way, we can access any index in essentially O(1) time on average. The time taken to search for the match in current user’s array is lessened and the time complexity of the algorithm improves by a factor of m.

9) In the end, we show the user those movies of the max\_similarity\_user which he or she hasn’t yet rated.

10)The time complexity of this method is O(n\*t) where n is the number of users and t is the average number of movies rated by each user.

**How does method 4 work?**

<insert a flow diagram for method 4>

1) This method is an improvement on method 3.

2) We don’t iterated over all the users in our database but only on those who have rated at least one such movie which has been rated by out current user.

3) This decreases the number of iterations on the user dataset and thus the time complexity is now lesser than O(n\*t) where n is the number of users and t is the average number of movies rated by each user.

4) To keep track of which users have at least one movie in common with our current user, we maintain a boolean array called relevant\_array and before we calculate the similarity scores, we fill this array.

5) To fill relevant\_array optimally, we use all the movies rated by the current user and find out which other users have rated these. For each such user, we set relevant\_user as true.

6) We are kind of taking the union of the users who have rated at least one movie rated by our current user.

7) There can be further improvements to this algorithm if the number of users increases by a lot. We can use kNN (k Nearest Neighbours) instead of iterating over all the users.