We Build this project during DSA Craft Hackathon, we are group of 3 member and we have to build solution in 5 hrs.

Our Problem Statement was to build backend system for social media platform. Which supports several features like analyzing user connections, recommending content, handling real-time events, and optimizing performance.

Sure, I'll explain the details focusing on how we implemented the solution and the specific data structures we used.

**### User Graph Analysis**

**\*\*Goal\*\*:** Analyze and manipulate the connections between users to find key users and groups, and develop features like friend recommendations.

**#### Data Structures Used:**

1. \*\*Adjacency List\*\*: Used to represent the user connections graph. This is an efficient way to store and traverse the graph.

- \*\*Data Structure\*\*: `unordered\_map<int, vector<int>>`

- \*\*Purpose\*\*: To store each user and their list of friends (connections).

2. \*\*Depth-First Search (DFS)\*\*: Algorithm used for traversing the user connections graph to find friend recommendations.

- \*\*Purpose\*\*: To explore connections up to a certain depth for recommending friends.

**### Implementation:**

1. \*\*Creating and Managing Connections\*\*:

- \*\*Adding Friends\*\*:

- We represent each user by a unique ID.

- When adding a friend, we update the adjacency list to reflect the new connection.

```cpp

static void addFriend(int user1, int user2) {

adjacencyList[user1].push\_back(user2);

adjacencyList[user2].push\_back(user1);

}

```

- \*\*Deleting Friends\*\*:

- We remove the friend from the adjacency list by erasing the connection.

```cpp

static void deleteFriend(int user1, int user2) {

auto& friends1 = adjacencyList[user1];

friends1.erase(remove(friends1.begin(), friends1.end(), user2), friends1.end());

auto& friends2 = adjacencyList[user2];

friends2.erase(remove(friends2.begin(), friends2.end(), user1), friends2.end());

}

```

**2. \*\*Analyzing Connections\*\*:**

- \*\*DFS Algorithm for Friend Recommendations\*\*:

- We use DFS to traverse the user connections graph up to a certain depth.

- This helps in finding and recommending friends based on existing connections.

```cpp

void dfs(int depth, int userId, unordered\_set<int>& visited, vector<User>& ans) {

if (depth == 0) return;

visited.insert(userId);

for (int friendId : Database::adjacencyList[userId]) {

if (visited.find(friendId) == visited.end()) {

ans.push\_back(Database::users[friendId]);

dfs(depth - 1, friendId, visited, ans);

}

}

}

```

**### Content Recommendation**

\*\*Goal\*\*: Suggest content to users based on their preferences and connections.

#### Data Structures Used:

1. \*\*Content\*\*: Represents individual pieces of content.

- \*\*Class\*\*: `Content`

- \*\*Attributes\*\*: `contentID`, `user`, `content`

2. \*\*User-Content Interaction Map\*\*: Maps user IDs to their interactions with content.

- \*\*Data Structure\*\*: `unordered\_map<int, vector<pair<int, int>>>`

- \*\*Purpose\*\*: To store and quickly access user interactions with content.

3. \*\*Content-User Interaction Map\*\*: Maps content IDs to user interactions.

- \*\*Data Structure\*\*: `unordered\_map<int, vector<pair<int, int>>>`

- \*\*Purpose\*\*: To store and quickly access which users interacted with a piece of content.

### Implementation:

1. \*\*Storing User-Content Interactions\*\*:

- When a user interacts with content, we update both the user-content interaction map and the content-user interaction map.

```cpp

static void addInteraction(int userId, Content& content, int interactionType) {

userContentMap[userId].push\_back({interactionType, content.contentID});

contentUserMap[content.contentID].push\_back({userId, interactionType});

}

```

**### Real-time Interaction**

\*\*Goal\*\*: Handle real-time events like friend requests and notifications efficiently.

#### Data Structures Used:

1. \*\*Request Queue\*\*: Stores incoming requests to be processed.

- \*\*Data Structure\*\*: `queue<Request>`

- \*\*Purpose\*\*: To manage and process requests in the order they arrive.

2. \*\*LRU Cache\*\*: Stores frequently accessed content for quick retrieval.

- \*\*Class\*\*: `LRUCache`

- \*\*Attributes\*\*: `capacity`, `cacheList`, `cacheMap`

- \*\*Purpose\*\*: To speed up access to frequently requested content.

### Implementation:

1. \*\*Handling Requests\*\*:

- Requests are added to a queue and processed in order.

```cpp

queue<Request> requestQueue;

void RequestHandler(Request r) {

if (r.type == 101) {

Database::addFriend(r.UserArr[0]->UserID, r.UserArr[1]->UserID);

} else if (r.type == 102) {

Database::deleteFriend(r.UserArr[0]->UserID, r.UserArr[1]->UserID);

}

// Other request types handled here

}

```

2. \*\*Caching Frequently Accessed Content\*\*:

- The LRU Cache ensures that the most frequently accessed content is quickly retrievable.

```cpp

class LRUCache {

private:

int capacity;

list<Content> cacheList;

unordered\_map<int, list<Content>::iterator> cacheMap;

public:

LRUCache(int capacity) : capacity(capacity) {}

string get(int contentID) {

if (cacheMap.find(contentID) == NULL) {

return "";

}

cacheList.splice(cacheList.begin(), cacheList, cacheMap[contentID]);

return cacheMap[contentID]->S;

}

void put(Content& content) {

if (cacheMap.find(content.contentID) != cacheMap.end()) {

cacheList.splice(cacheList.begin(), cacheList, cacheMap[content.contentID]);

cacheMap[content.contentID] = cacheList.begin();

return;

}

if (cacheList.size() == capacity) {

int lastContentID = cacheList.back().contentID;

cacheList.pop\_back();

cacheMap.erase(lastContentID);

}

cacheList.push\_front(content);

cacheMap[content.contentID] = cacheList.begin();

}

};

```

**### Performance Optimization**

\*\*Goal\*\*: Ensure the system runs efficiently and can handle high loads.

#### Techniques Used:

1. \*\*Caching\*\*: Storing frequently accessed data in a fast, temporary storage.

- \*\*Implemented using\*\*: `LRUCache`

2. \*\*Database Indexing\*\*: Ensuring quick access to important data fields.

- \*\*Simulated using\*\*: Efficient data structures like hash maps.

**### Overall System Architecture**

\*\*Servers and Load Balancing\*\*:

1. \*\*Server Class\*\*: Each server handles requests and manages a cache.

```cpp

class Server {

public:

int serverID;

int activeConnections;

queue<Request> requestQueue;

LRUCache cache;

Server(int id, size\_t cacheCapacity) : serverID(id), activeConnections(0), cache(cacheCapacity) {}

void RequestHandler(Request r) {

// Handle the request

}

};

```

2. \*\*System Class\*\*: Manages multiple servers and balances the load using consistent hashing.

```cpp

class System {

private:

vector<Server> servers;

// Other attributes

public:

void AddServer(size\_t cacheSize) {

servers.emplace\_back(servers.size(), cacheSize);

}

void AddRequestToQueue(Request req) {

// Distribute request to appropriate server

}

void ProcessQueue() {

// Process requests on each server

}

};

```

**Challenges and Learnings::**

**This was our first hackthon in which we were going to work on DSA base project even though we have good knowledge of DSA and oops concept but we don’t have experience at that time so we started working indivisully on diff functionality but we realize that code merging will take more time as everyone is approaching subproblem on their own way so we decide to firstly design a proper design that what we have to integret and how we are going to implement.so we apply all our knowledge of oops concept and create a class decide class member such that at the end we will able to merge code efficiently.  
  
int main() {**

**System system;**

**system.AddServer(10); // Adding a server with cache size 10**

**system.AddServer(50); // Adding another server with cache size 50**

**User user1; // Creating user1**

**User user2; // Creating user2**

**User user3; // Creating user3**

**// user1 sends friend requests to user2 and user3**

**Request req = user1.CreateRequest(user2, false); // Add friend**

**Request req1 = user1.CreateRequest(user3, false); // Add friend**

**// Adding requests to the system queue**

**system.AddRequestToQueue(req);**

**system.AddRequestToQueue(req1);**

**// Processing the requests**

**system.ProcessQueue();**

**// Displaying the adjacency list (friends)**

**for(auto it : Database::adjacencyList) {**

**cout << it.first << ": ";**

**for(auto it2 : it.second) {**

**cout << it2 << " ";**

**}**

**cout << endl;**

**}**

**// user1 deletes user3 as a friend**

**Request req3 = user1.CreateRequest(user3, true); // Delete friend**

**system.AddRequestToQueue(req3);**

**system.ProcessQueue();**

**// Displaying the updated adjacency list**

**for(auto it : Database::adjacencyList) {**

**cout << it.first << ": ";**

**for(auto it2 : it.second) {**

**cout << it2 << " ";**

**}**

**cout << endl;**

**}**

**return 0;**

**}**