**Design Documentation**

This document outlines the database schema design for a dating application.

**SQL Approach**

**1. User Profiles**

**Table**: UserProfiles

**Fields**:

UserID (INT, PRIMARY KEY): Unique identifier for the user.

Username (VARCHAR(50), UNIQUE): Username for login.

Email (VARCHAR(100), UNIQUE): User's email address.

PasswordHash (VARCHAR(255)): Hashed password for secure authentication.

Gender (ENUM): User's gender ('Male', 'Female', 'Other').

DateOfBirth (DATE): User's date of birth.

Latitude (POINT): User's location stored as a geographical point (latitude).

Longitude (POINT): User's location stored as a geographical point (longitude).

Bio (TEXT): User's biography/description.

ProfilePictureURL (VARCHAR(255)): URL of the user's profile picture (optional).

**Rationale**:

User ID as primary key for efficient data retrieval.

Unique username and email for user identification and login.

Password stored securely using a hash and salt combination.

Gender stored as a reference to a separate Genders table for flexibility and future expansion.

Latitude and longitude stored as a point for geospatial search functionalities.

Bio and profile picture URL for user self-expression.

**2. User Preferences**

**Table**: UserPreferences

**Fields**:

UserID (INT, PRIMARY KEY, FOREIGN KEY REFERENCES UserProfiles(UserID)): Unique identifier for the user (references UserProfiles table).

PreferredMinAge (INT): Minimum age preference for matches.

PreferredMaxAge (INT): Maximum age preference for matches.

PreferredGenders (VARCHAR(255)): Comma-separated list of genders

DistanceRadius (INT): Maximum search radius for matches in kilometers.

NotificationSettings (BOOLEAN): User's notification preferences stored as true or false.

**Rationale**:

One-to-one relationship with Users table.

Separate table for user preferences for better organization and scalability.

Age range for filtering matches.

Comma separated list of genders for flexible preference selection (e.g., multiple genders).

Distance radius for location-based matching.

Boolean format for storing various notification settings.

**3. Matches**

**Table**: Matches

**Fields**:

MatchID (INT, PRIMARY KEY): Unique identifier for the match.

UserID1 (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the first person.

UserID2 (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the second person.

MatchStatus (ENUM('Pending', 'Accepted', 'Declined')): Status of the match (pending, accepted, or declined).

MatchDateTime (DATETIME): Timestamp of the match creation.

**Rationale**:

Many-to-many relationship between Users through Matches table.

Efficient indexing on user IDs for quick retrieval of matches.

Status for tracking match progress.

Timestamp for recording match creation time.

**4. Messages**

**Table**: Messages

**Fields**:

MessageID (INT, PRIMARY KEY): Unique identifier for the message.

SenderID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the sender.

ReceiverID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the receiver.

MessageContent (TEXT): Text content of the message.

Timestamp (DATETIME): Timestamp of the message sent.

ReadStatus (BOOLEAN): Flag indicating if the message is read.

Attachments (VARCHAR(255)): URL of any attached media (optional).

**Rationale**:

Many-to-many relationship between Users through Messages table.

Indexing on sender/receiver IDs and timestamp for optimized message retrieval.

Flag for tracking message read status.

Optional attachment URL for media attachments.

**5. Likes/Dislikes**

**Table**: LikesDislikes

**Fields**:

LikeID (INT, PRIMARY KEY): Unique identifier for the like/dislike action.

LikerID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the liking person.

LikedUserID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the liked/disliked person.

LikeStatus (ENUM('Like', 'Dislike')): Type of action (like or dislike).

Timestamp (DATETIME): Timestamp of the like/dislike action.

**Rationale**:

One-to-many relationship between Users and Likes tables (a user can like/dislike many profiles).

Foreign keys ensure data integrity.

Separate like\_status for clear indication of user preference.

Timestamp for tracking like/dislike activity.

**Constraints**:

UNIQUE (LikerID, LikedUserID): This unique constraint prevents duplicate likes/dislikes from the same user on the same profile.

**6. Reporting and Moderation (Optional)**

**Table**: Reports

**Fields**:

ReportID (INT, PRIMARY KEY): Unique identifier for the report.

ReporterID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the reporter.

ReportedUserID (INT, FOREIGN KEY REFERENCES UserProfiles(UserID)): User ID of the reported profile.

ReportedMessageID (INT, FOREIGN KEY REFERENCES Messages(MessageID)): ID of the reported message (optional).

ReportReason (TEXT): Reason for reporting the user/message.

Timestamp (DATETIME): Timestamp of the report creation.

ModerationAction (ENUM('Pending', 'Warning', 'Suspension')): Moderation action taken (pending, warning, or suspension).

**Rationale**:

Optional tables for reporting and moderation functionalities.

Relationships with Users and Messages tables for data association.

Fields for capturing report details, reason, actions, and involved users.

This schema design prioritizes data integrity, scalability, and efficient retrieval of information for the dating app.

\* **Normalization**: Normalization techniques (1NF, 2NF) are applied to minimize data redundancy. Further normalization could be achieved based on actual data usage patterns and performance considerations.

\* **Foreign Keys**: Foreign keys ensure referential integrity between tables.

\* **Efficiency**: Indexing on relevant fields like UserID, location(Latitude, Longitude), timestamps facilitates fast facilitates fast retrieval of data for functionalities like searching for matches, displaying profiles, and managing messages.

\* **Scalability**: The schema can handle a large number of users and messages efficiently. However, sharding or partitioning strategies may need to be considered for extremely large datasets.

\* **Future Enhancements**: The schema can be extended to incorporate additional features as the dating app evolves (e.g., adding interests, adding voice/video chat functionalities, chat room functionality or more sophisticated matching algorithms could be considered for future enhancements).

**Security Considerations**

\* Secure password hashing should be implemented using industry-standard algorithms (e.g., bcrypt, scrypt).

\* User data like email and location should be stored securely and only accessed with proper authorization.

\* Regular security audits and penetration testing are recommended to identify and address vulnerabilities.

With this design, the database can efficiently handle user profiles, preferences, matches, messages, likes/dislikes, and reporting/moderation functionalities for the dating app.

**NoSQL Approach**

**Schema Design**

**1. User Profiles Table**:

- Partition Key: UserID (String)

- Attributes:

- Username (String)

- Email (String)

- PasswordHash (String)

- Gender (String)

- DateOfBirth (String or Number)

- Location (Map): {Latitude: Number, Longitude: Number}

- Bio (String)

- ProfilePictureURL (String)

- Secondary Indexes: None

**2. User Preferences Table:**

- Partition Key: UserID (String)

- Attributes:

- PreferredMinAge (Number)

- PreferredMaxAge (Number)

- PreferredGenders (String Set)

- DistanceRadius (Number)

- NotificationSettings (Boolean)

- Secondary Indexes: None

**3. Matches Table:**

- Partition Key: MatchID (String or Number)

- Attributes:

- UserID1 (String)

- UserID2 (String)

- MatchStatus (String)

- MatchDateTime (String or Number)

- Secondary Indexes: None

**4. Messages Table:**

- Partition Key: ConversationID (String)

- Sort Key: Timestamp (String or Number)

- Attributes:

- SenderID (String)

- ReceiverID (String)

- MessageContent (String)

- ReadStatus (Boolean)

- Attachments (String Set)

- Secondary Indexes: None

**5. Likes/Dislikes Table:**

- Partition Key: UserID (String)

- Sort Key: LikedUserID (String)

- Attributes:

- LikeStatus (String)

- Timestamp (String or Number)

- Secondary Indexes: None

**6. Reporting/Moderation Table:**

- Partition Key: ReportID (String or Number)

- Attributes:

- ReporterID (String)

- ReportedUserID (String)

- ReportedMessageID (String)

- ReportReason (String)

- ModerationAction (String)

- Timestamp (String or Number)

- Secondary Indexes: None

**Rationale and Documentation:**

In DynamoDB, we denormalize data to minimize the need for joins and optimize for read operations. Each table is designed to serve specific query patterns, and secondary indexes can be added later if needed for additional query flexibility.

\* **Normalization**: DynamoDB doesn't follow the normalization principles of SQL databases. Instead, data is structured based on access patterns and query requirements.

\* **Efficiency**: DynamoDB offers high performance and scalability out of the box, with automatic scaling and low-latency reads and writes. Designing tables based on query patterns helps optimize performance.

\* **Scalability**: DynamoDB scales horizontally to handle large amounts of data and high throughput. As the application grows, DynamoDB can handle increased load with minimal management overhead.

\* **Future Enhancements**: Additional features like real-time notifications, geospatial queries, or machine learning-based matchmaking algorithms can be implemented with DynamoDB streams, AWS Lambda, and other AWS services.

**Data Access**:

\* DynamoDB uses document model for data, allowing flexible schema updates without altering the entire table structure.

\* Primary and composite keys are crucial for efficient data retrieval based on specific user IDs, message IDs, etc.

\* Secondary Indexes can be created on frequently queried attributes (e.g., gender\_preference\_ids) for faster lookups.

**Security Considerations**:

\* Implement strong password hashing algorithms like bcrypt or scrypt.

\* Use IAM policies to control access to DynamoDB tables and attributes based on user roles. This ensures that only authorized users can access and modify specific data. For example, a basic user might only have read access to their profile information, while moderators might have access to report data and take moderation actions.

\* Encrypt sensitive data at rest and in transit using AWS Key Management Service (KMS) to further protect user information.

\* Regularly monitor and audit your DynamoDB tables for any suspicious activity.

**Benefits of NoSQL for Dating App**:

\* **Scalability**: NoSQL databases like DynamoDB can easily scale horizontally to handle increasing user base and data volume.

\* **Performance**: DynamoDB offers fast data access due to its distributed architecture and focus on specific use cases like key-value retrieval.

\* **Flexibility**: The schema-less nature of DynamoDB allows for flexible data modeling and easier adaptation to evolving app features.

Overall, DynamoDB provides a flexible and scalable foundation for building the database schema of dating app, catering to the requirements of modern, high-traffic applications.

Utilizing DynamoDB's features like primary/composite keys, secondary indexes, and access control mechanisms allows for efficient data management and retrieval while prioritizing security through strong password hashing, IAM policies, and data encryption. By leveraging the scalability and performance benefits of NoSQL, this approach can support dating app with a robust and secure data foundation.