Bitchip

Texas Hold ‘em Poker Backend Design

Group 12
Jonathan Benson, Kevin Floyd, Joshua Newton, Jacob Smith, Thomas Armenta
5 May 2022
Final Report

Project Git Repository
https://github.com/jonathanbenson/Texas-Holdem-MySQL

Introduction:

Texas Hold 'em is arguably the most popular variation of the classic card-game poker. The game consists of 2-10 players who buy their way into a table. At the start of each round, two players are forced to bet the big and small blinds. Each player then draws two random cards into their hand from a standard 52-set card deck. Their hands are supposed to be kept secret from the rest of the players. The player immediately clockwise to the big blind is the one first to play their turn. They can either call, raise, or fold. A call is when a player bets the same amount the player bet before them. A raise is when a player raises the current bet greater than the amount of the previous player. And finally, a fold is when a player returns their cards to the dealer to avoid potentially betting more chips that they might lose later to a stronger hand.

Each time a player bets, their bet is added to the pot. If all players but one fold their cards, the remaining player receives the pot as a reward. Otherwise, three cards are *flop*ped randomly onto the middle of the table for everyone to see. Another round of betting then ensues; this time they have the ability to check. A player can check if no previous players bet a raise. Once all players have either checked, called, or folded, then a fourth card is *turn*ed onto the table. This process repeats once more, and finally a fifth card is thrown into the *river*. After a final round of betting ensues, each player’s card is revealed for a final *showdown*. The player with the highest quality hand with respect to the community cards receives the pot as a reward. Then another match begins, and the small and big blinds are rotated clockwise by one player.

We will be designing a backend system that allows users to buy into a variety of virtual poker tables and play poker with one another remotely. The RDMS which was used was MySQL 8.0.

Division of Labor

The following division of labor expresses what we as team members were responsible for completing, but all team members frequently reached out to help each other on various occasions.

- Jonathan Benson
    - Facilitated changes to the git repository by ensuring adequate testing, writing proper documentation, and occasionally resolving merge conflicts.
    - Authored the CLEAN, NEW_TABLE, NEW_MATCH, and SHUFFLE_DECK stored procedures
    - Authored the Final Report's Database Model Features
    - Authored the majority of the SQL schema
- Jacob Smith
    - Helped design, write, and test the SQL schema and stored procedures.
    - Authored the LEAVE_TABLE and GET_EMPTY_TABLE_SEAT stored procedures
- Kevin Floyd
    - Designed the majority of the ER diagram and made sure it matched the SQL schema
    - Authored the Final Report's Introduction
- Thomas Armenta
    - Handled setup, management, and proofreading of the milestones and final report.
    - Authored the Final Report's Application Requirements, Database Requirements, and part of the Future Scope
    - Contributed ideas to the design of the database model and stored procedures.
- Joshua Newton
    - Helped design and document the SQL schema and stored procedures.
    - Authored the JOIN_TABLE stored procedure
    - Authored the Final Report's Deficiencies section, and also made large contributions to the Future Scope
    - Aided in proofreading the Final Report

Database Requirements

1. User:
    a. A User will have a unique username up to 255 characters as a primary key (ID)
    b. A User will have a password
    c. A User will have a purse (the amount of chips they have).
2. Card:
    a. A Card will have a Face from a list of 13 possible options
    b. A Card will have a Suit from a list of 4 possible options
3. Deck:
    a. A Deck will start with a total of 52 records
    b. A Deck will have a Face match with each Suit for all 52 possibilities, where no records are the same
4. Hand:
    a. A Hand will have a system generated primary key (ID)
    b. A Hand will have only two Hand Card ID's dealt to each player from the Deck
    c. A Hand will have the username of the player/user
5. Community Card:
    a. A Community Card will record the current Round ID
    b. A Community Card will have a Face and Suit from the Deck
6. Hand Type:
    a. A Hand Type will have a Rank as a primary key (ID)
    b. A Hand Type will have a Name
    c. A Hand Type will have a Degree integer to be compared to other values
7. Hand Type Instance:
    a. A Hand Type Instance will have a system generated primary key (ID)
    b. A Hand Type Instance will have a Hand Type Rank integer from the Hand Type degree
8. Table:
    a. A Table will have a system generated primary key (ID)
    b. A Table will have 10 seats
    c. A Table will have 2 - 10 users that can join in between Matches.
    d. A Table will have one big blind
    e. A Table will have one small blind
    f. A Table will have a buy-in amount that is x4 the small blind
9. Seats:
    a. A Seat will have a system generated primary key (ID)
    b. A Seat will have a Sitter Username that tells who is occupying the seat (NULL in the case of nobody)
    c. A Seat will have the Table Id it is placed at

10. Match:
    a. A Match will have a system generated primary key (ID)
    b. A Match will have a Table ID from where it was created
    c. A Match will have a Winner as the Users username (NULL when the match is underway)
    d. A Match will have the last Match ID from the last match played
    e. A Match will have a pot which holds the amount of chips from bets
11. Round:
    a. A Round will have a system generated primary key (ID)
    b. A Round will have the Match Id from where it was generated
    c. A Round will have a Betting Round from a list of four possible types of betting: the pre-flop, flop, turn, and river.
12. Play:
    a. A Play will have a system generated primary key (ID)
    b. A Play will have the Username from the user it was generated
    c. A Play will have the Round Id from the round it was generated
    d. A Play will have a Play Type from a list of 4 Possible voluntary types: Check, Call, Raise, or Fold, or be of an involuntary type: small or big blind
    e. A Play will have the Last Play Id
    f. A Play will have a Chip Number to use as a bet

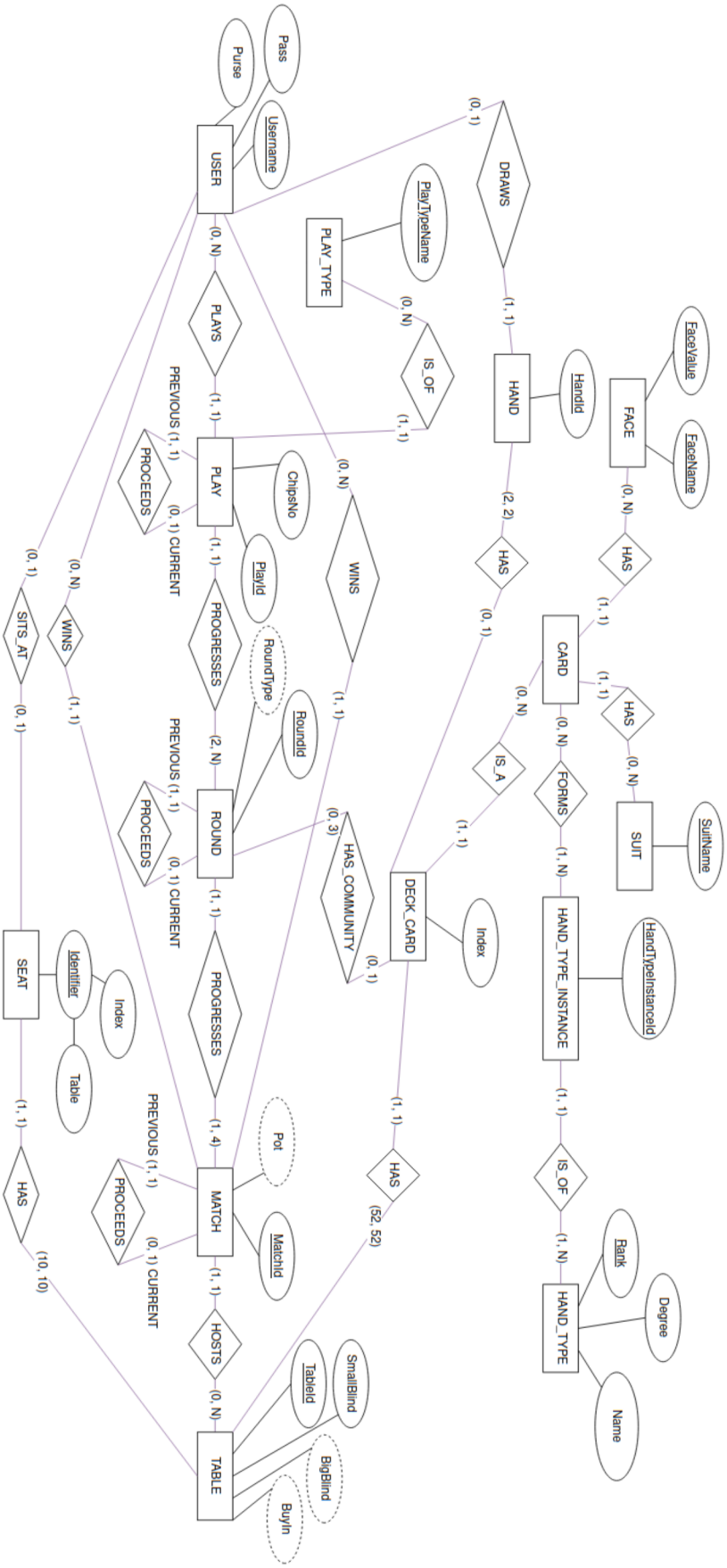
Application Requirements

Due to the complexity of the database model's business logic, we did not implement a full-stack application for our project. If we were to implement one in the future, here are some features we would make sure to be incorporated in the finished piece:

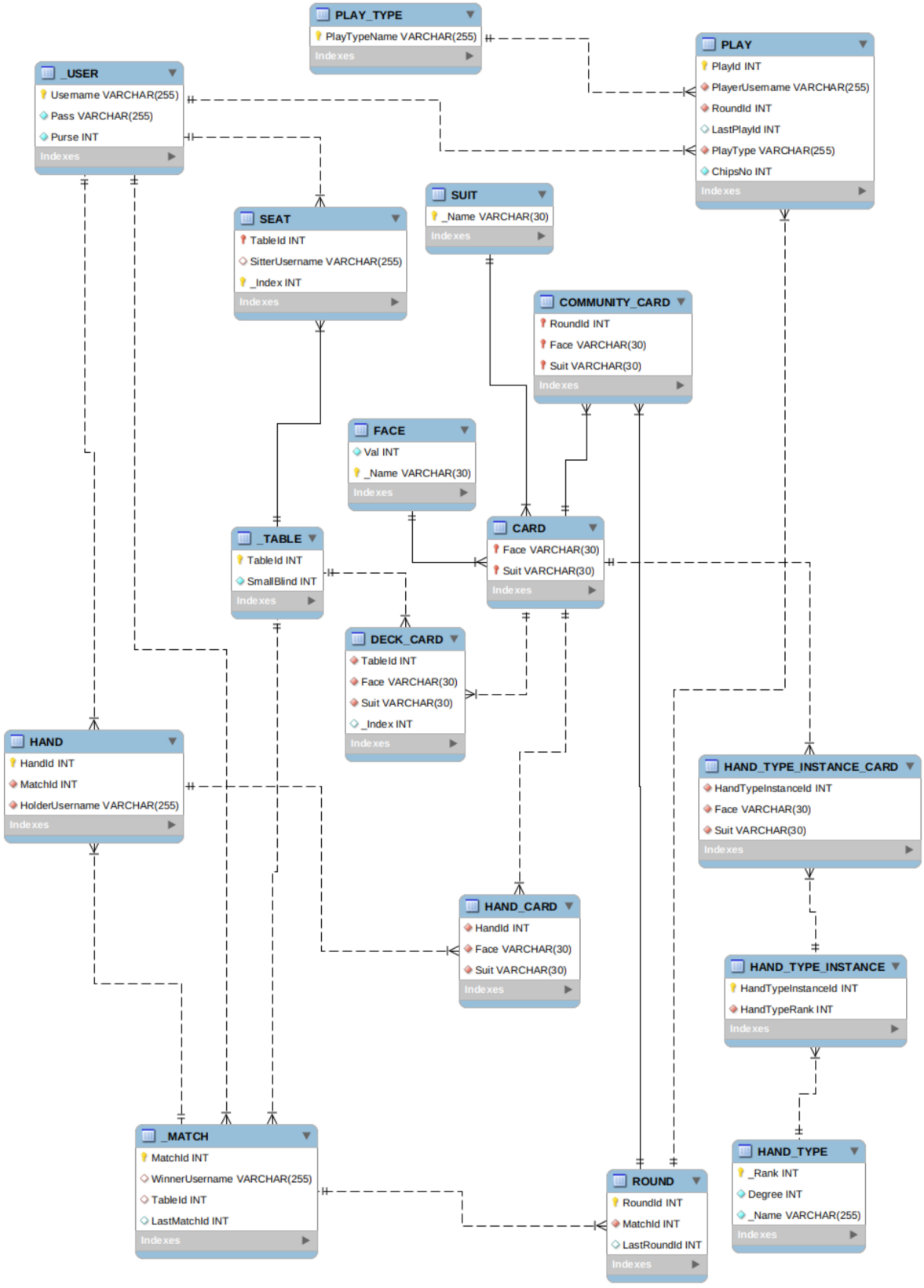
1. Security
    a. SQL injection would be prevented mainly through stored procedures, and provide a few views for simple read-only operations.
    b. To authenticate our admins, we would make it so that they can only access the database from a specific machine.
    c. Implement Role Based Access Control for each type of application that needs access to the database.
    d. To prevent DDOS attacks we would outsource to a company like Cloudflare for their SaaS.
2. User:
    a. A User will have to create an account to set up username and password
    b. Application should be able to add chips to the Purse
3. Card:
    a. A Card should have a reveal side and a hidden side
4. Deck:
    a. A Deck should be shuffled at the start of a Match
    b. A Deck should be decremented as Cards are handed out
    c. A Deck should not allow Cards to be added back during a Match
5. Hand:
    a. A Hand is dealt to each User, after the shuffle, in a clockwise linear fashion
    b. Each Player should atomically hide their hand
    c. A Hand should be able to fold during Play
6. Community Card:
    a. Three Community Cards will be dealt from the deck after the first betting round
    b. A Community Card will be revealed starting the next betting round
    c. A maximum of 5 Community Cards. The first 3 are known as the flop, the fourth the turn, and the fifth the river.
7. Hand Type Instance:
    a. A Hand Type Instance will be used to determine a winner
    b. Each Hand Type Instance has a rank and a reference to a six-card possibility
8. Table:
    a. Should only start a Match with a minimum of 4 Users and max of 10
    b. A table should give users who played in the previous match priority over users who are waiting to join

9. Match:
    a. The Match will increment the Pot after each Betting round from the Users who Played
    b. A Match will have Current Number Users, starting with the total Users at the Table and decimating if Users Fold
    c. The winner of the Match will be awarded the amount in the Pot
    d. A Match can end early if all but one user folds
    e. After a Match is over the big blind and little blind will shift to the left/clockwise one seat
10. Round:
    a. Before the start of the first pre-flop betting the big blind and little blind users must pay to play
    b. A Round of Play/Betting will start with the User to the left/Clockwise of the big blind
    c. All rounds after the first will have the little blind start the betting or the first active user clockwise from little blind
    d. A Round should not end until all active users have the same bet amount
11. Play:
    a. A Play should decrement the amount of Chips bet from the Users purse
    b. A Play should determine the choices a User has based on amount of Chips in purse and the previous play
    c. A Play is created by a user, each going one at a time in a clockwise linear fashion
    d. Each player would be granted a limited amount of time to decide, or else automatically fold their hand.

ER Diagram


FaceValue
FaceName
FACE
HandId
HAS
(0, N)
(1, 1)
CARD
HAS
SUIT
SuitName
HandTypeInstanceId
HAND_TYPE_INSTANCE
IS_OF
HAND_TYPE
Rank
Degree
Name
FORMS
IS_A
DECK_CARD
Index
HAS
(52, 52)
DRAWS
HAND
(2, 2)
HAS_COMMUNITY
(0, 3)
Pass
Purse
Username
USER
PlayTypeName
PLAY_TYPE
IS_OF
PLAYS
PLAY
PROCEEDS
PREVIOUS (1, 1)
(0, 1) CURRENT
ChipsNo
PlayId
WINS
PROGRESSES
RoundType
RoundId
ROUND
SITS_AT
SEAT
Identifier
Table
Index
Pot
MatchId
MATCH
HOSTS
TableId
SmallBlind
BigBlind
BuyIn
TABLE
(10, 10)

Schema Diagram



Database Model Features

The following showcase our most important stored procedures that implement a significant portion of our backend's functionality.

Creating a New Table

```
CREATE PROCEDURE NEW_TABLE (IN smallBlind INT)
BEGIN
    /*
    Creates a new table and initializes its seats and deck of cards
    */

    -- Variable used for the newly created table id
    DECLARE newTableId INT DEFAULT 0;

    -- Variables used for initializing the cards in the table's deck
    DECLARE count INT DEFAULT 1;
    DECLARE currentFace VARCHAR(30) DEFAULT "";
    DECLARE currentSuit VARCHAR(30) DEFAULT "";

    -- Insert new table into database with given small blind
    INSERT INTO _TABLE (SmallBlind) VALUES (smallBlind);

    -- Retrieve the table id of the newly created table
    SELECT MAX(TableId) INTO newTableId
    FROM _TABLE
    GROUP BY TableId
    LIMIT 1;

    -- Initialize the 10 seats for the new table
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 0);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 1);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 2);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 3);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 4);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 5);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 6);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 7);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 8);
    INSERT INTO SEAT (TableId, _Index) VALUES (newTableId, 9);

    -- Initialize cards in the table's deck
    WHILE count <= 52 DO

        SELECT Face INTO currentFace
        FROM CARD
        WHERE Id = count
        LIMIT 1;

        SELECT Suit INTO currentSuit
        FROM CARD
        WHERE Id = count
        LIMIT 1;

        INSERT INTO DECK_CARD (TableId, Face, Suit, _Index)
        VALUES (newTableId, currentFace, currentSuit, count);

        SET count = count + 1;

    END WHILE;

END $$
```

Poker tables are the foundation of our backend system. To initialize a new table, several operations must be performed. We implemented the logic associated with creating a new table in the NEW_TABLE stored procedure. It receives only one argument: the small blind of the new table.

First, a new table record must be inserted into the database requiring only information about the small blind. Second, 10 seats must be created for the table in order for players to join. Finally, the table's deck of cards must be created.

A deck of cards in our backend system is simply a set of 52 poker cards that have unique indices. Their indices specify their placement in the deck. We initialize the decks by inserting copies of CARD records one-by-one into the DECK_CARD table; each with their own unique index.

Sitting at a Table

```
CREATE PROCEDURE JOIN_TABLE (In playerName varchar(255), In pokerTable INT, OUT msg Varchar(100))
BEGIN

    SET @playerId = NULL;
    SET @seatId = NULL;
    SET @smallBlind = NULL;
    SET @purse = NULL;

    # fetch playerId of that player from player table using player name or
    # else store player name directly into the pokerTable
    SELECT Username INTO @playerId
    FROM _USER
    WHERE Username=playerName;

    IF(@playerId IS NULL)THEN
    BEGIN
        SET msg='FAIL - USER NOT FOUND';
    END;
    ELSE
    BEGIN

        -- Find out the small blind of the given table - used to calculate the buy-in
        SELECT SmallBlind INTO @smallBlind
        FROM _TABLE
        WHERE TableId = pokerTable;

        -- Find out the # chips the user has in their purse
        SELECT Purse INTO @purse
        FROM _USER
        WHERE Username = @playerId;

        -- If the user has less chips in their purse than the amount of the buy-in (x4 the small blind)
        -- then they cannot sit at the table
        IF (@purse < (@smallBlind * 4)) THEN
        BEGIN
            SET msg = 'FAIL - NOT ENOUGH FUNDS';
        END;
        ELSE
        BEGIN

            # check that player present in that table or not
            SELECT _Index INTO @seatId
            FROM SEAT
            INNER JOIN _TABLE ON _TABLE.TableId=pokerTable
            WHERE SEAT.SitterUsername=@playerId
            LIMIT 1;

            # if seat id is null it means that player have no seat in that table
            # then check is there any seats empty
            IF(@seatId IS NULL)THEN
            BEGIN

                SELECT _Index INTO @seatId
                FROM SEAT
                INNER JOIN _TABLE ON _TABLE.TableId=pokerTable
                WHERE SEAT.SitterUsername IS NULL
                LIMIT 1;

                # if seat id is not null it means empty seat is present
                # then give that seat to that player
                IF(@seatId IS NOT NULL)THEN
                BEGIN
                    UPDATE SEAT
                    SET SitterUsername=@playerId
                    WHERE _Index=@seatId AND TableId=pokerTable;

                    SET msg='SUCCESS';
                END;
                ELSE
                BEGIN
                SET msg='FAIL - NO EMPTY SEATS';
                END;
                END IF;
            END;
            ELSE
            BEGIN
                SET msg='FAIL - PLAYER ALREADY AT TABLE';
            END  ;
            END IF;
        END;
        END IF;
    END;
    END IF;
END $$
```

Before a user can start playing poker, they have to join a table. We implemented the JOIN_TABLE stored procedure to handle this logic. It receives three arguments: the username of the player wanting to join, the poker table, and an output status message.

First, it checks whether or not the user intending to join actually exists in the database. Second, it checks if the user has enough chips in their purse to afford the buy-in of the table (x4 the small blind). Finally, the procedure checks if there are any seats available. In case of any difficulty joining the table, we implemented a message parameter that will tell the user why they may not be allowed to join.

If all of the above conditions are met, then the JOIN_TABLE procedure will select the index of the first available seat and place the user at the table.

Beginning a New Match

```
CREATE PROCEDURE NEW_MATCH (IN tableId INT, OUT msg VARCHAR(100))
BEGIN
    /*
    Creates a new poker match at a given table.
    */

    DECLARE numPlayers INT DEFAULT 0;

    DECLARE lastMatch INT DEFAULT NULL;

    -- Find out how many players are sitting at the table
    SELECT COUNT(SitterUsername) INTO numPlayers
    FROM SEAT
    GROUP BY TableId
    HAVING TableId = tableId
    LIMIT 1;

    -- If there are less than 4 players sitting at the table
    -- then we cannot begin a new match.
    SET msg = CASE
        WHEN numPlayers < 4 THEN "FAIL - NOT ENOUGH PLAYERS"
        ELSE "SUCCESS"
    END;

    IF (msg = "SUCCESS") THEN
    BEGIN

        -- Find out the most recent match before this one
        SELECT MatchId INTO lastMatch
        FROM _MATCH
        WHERE TableId = tableId AND MatchId IS NOT NULL
        ORDER BY MatchId DESC
        LIMIT 1;

        -- Insert a new match into the database
        -- with the most recent match as its last match
        INSERT INTO _MATCH (TableId, LastMatchId)
        VALUES (tableId, lastMatch);

    END;
    END IF;

END $$
```

Beginning a new match is not as simple as just inserting a single new _MATCH record into the database. We implemented the logic associated with creating a new match in the NEW_MATCH stored procedure. It takes two arguments: the poker table, and an output status message.

A match cannot begin without at least 4 players waiting at the table. We check how many players are sitting at the table by counting the number of seats at the table whose SitterUsername is not NULL.

We designed our _MATCH table with a special LastMatchId attribute. It behaves similar to a linked-list, but with SQL circular references. Accessing the previous match is important because the blind betters must be rotated clockwise by 1 player in the new match. Without knowing the blind betters of the previous match, the new blind betters could be the same as the ones in the previous match.

Shuffling a Deck of Cards

```sql
CREATE PROCEDURE SHUFFLE_DECK (IN tableId INT)
BEGIN
    /*
    Shuffles a table's deck of cards.
    */

    DECLARE numShuffles INT DEFAULT 0;

    DECLARE indexA INT DEFAULT NULL;
    DECLARE faceA VARCHAR(30) DEFAULT "ACE";
    DECLARE suitA VARCHAR(30) DEFAULT "SPADES";

    DECLARE indexB INT DEFAULT NULL;
    DECLARE faceB VARCHAR(30) DEFAULT "2";
    DECLARE suitB VARCHAR(30) DEFAULT "CLUBS";

    -- Swap two cards' indices in the deck 104 times (two times length of deck)
    WHILE numShuffles < 104 DO

        -- Retrieve the current indices of the two cards we want to swap
        SELECT _Index INTO indexA
        FROM DECK_CARD
        WHERE TableId = tableId AND Face = faceA AND Suit = suitA;

        SELECT _Index INTO indexB
        FROM DECK_CARD
        WHERE TableId = tableId AND Face = faceB AND Suit = suitB;

        -- Swap the two cards' indices
        UPDATE DECK_CARD
        SET _Index = indexA
        WHERE TableId = tableId AND Face = faceB AND Suit = suitB;

        UPDATE DECK_CARD
        SET _Index = indexB
        WHERE TableId = tableId AND Face = faceA AND Suit = suitA;

        -- Generate two random cards' face and suit to swap next
        -- ...using indexA and indexB variables for convenience here
        SET indexA = FLOOR(RAND() * 52) + 1;
        SET indexB = FLOOR(RAND() * 52) + 1;

        SELECT Face, Suit INTO faceA, suitA
        FROM CARD
        WHERE Id = indexA;

        SELECT Face, Suit INTO faceB, suitB
        FROM CARD
        WHERE Id = indexB;

        SET numShuffles = numShuffles + 1;

    END WHILE;

END $$
```

Poker would not be considered gambling if the decks were not randomly shuffled! We implemented the shuffling algorithm in our SHUFFLE_DECK stored procedure. It digests only one argument: the parent poker table.

All DECK_CARD records in our backend system are unique by the combination of their _Index and TableId attributes. Our approach to shuffling was then to simply shuffle all the _Index attributes of DECK_CARD records whose TableId equaled the one provided to the procedure call.

We implemented the SHUFFLE_DECK algorithm in code by utilizing the handy built-in RAND() function. The procedure swaps two randomly selected DECK_CARD records' _Index value 104 times. We figured 104 was a good number of swaps for the shuffle because it is enough to ensure randomness, but not too excessive to lower performance.

Future Scope

Front-End

Adding a front-end to complement the functions of the database and backend would be the next major step. The application would need a login/create account feature for new users. A home screen where users can add funds or change login credentials would also be necessary. Additionally, a lobby room with tables and their available seats would allow users to join tables. Users would play poker at a virtual poker table where they could see the usernames, avatars, and bets of other players.

Scaling

Setting our application up to run online to reach a bigger audience will take additional application adjustments. For a small to medium size application we would be fine using our original MySQ-only backend. Once the user-base grows larger, however, we would most likely need to shard our MySQL database to balance the load. Horizontal scaling with NoSQL databases like MongoDB and Cassandra may also provide additional performance and storage benefits.

Limits of the MySQL API

It is important to recognize that although MySQL is technically turing-complete through its API, it lacks the ability to implement external I/O operations. If we were to scale with other data-storage technologies we would need to implement an additional business-logic layer to handle I/O operations outside of MySQL.

Additional Backend Features

- Sometimes players become unresponsive after a while, so we would implement a timer that automatically kicks players from matches to discourage inactivity.
- Players overtime may improve their poker skills. A ranking system to group players based on their earnings could prevent experienced poker players from preying on the noobs.
- Crypto has been all the rage in recent years. We could implement tables where players could bet with cryptocurrencies like Bitcoin, Ethereum, and Dogecoin.

Deficiencies

- No front-end application for a user to interact with, that is connected to the database.
- The database may need some more tables regarding account creation on an application. We could have created a procedure to automatically generate new tables when needed and close out old tables when less players are online.
- No procedure to determine the winning hand in a poker match
- No implementation for the creation of rounds and plays