Brief description of the AiDex protocol

Table of contents

1. Introduction

Purpose of the Document

This document outlines the functionality and interactions of the automated market-making (AMM) protocol with concentrated liquidity implemented on the Solana blockchain. It serves as a comprehensive guide for auditors, detailing the system's components, workflows, and expected behaviors to facilitate a thorough evaluation of its operational integrity and security.

High-Level System Architecture

The protocol employs a modular and scalable architecture, emphasizing clarity and separation of responsibilities across its components. Key elements include:

- **Administrative Layer**: Governs system-wide configurations and updates through admin accounts and configuration accounts.
- **Liquidity Pools**: Core components for executing swaps and managing liquidity. These pools are initialized with specific parameters, including fee tiers and tick arrays, to optimize trading and liquidity allocation.
- **Position Management**: Enables LPs to create, track, and adjust liquidity positions within defined price ranges.
- **Fee System**: Implements a multi-tiered structure for protocol and LP fees, ensuring equitable distribution of rewards while maintaining competitiveness in the DeFi market.

The integration of innovative features, such as temporary liquidity pools, referral mechanisms for LPs and swaps, and automatic reinvestment strategies, further distinguishes the protocol. These enhancements aim to provide a dynamic, user-friendly experience while maximizing returns for LPs and ensuring robust market activity.

2. Protocol Components

2.1 Administration Layer

The administration layer provides the foundation for managing and configuring the protocol, ensuring robust governance and flexibility. It encompasses two primary components: Super Admin Account and Configuration Accounts.

2.1.1 Super Admin Accounts

The Super Admin Account is the central authority within the protocol, designed to manage critical administrative operations securely. It is implemented using the one unique Program Derived Address (PDA) mechanism provided by the Solana blockchain. This design prevents the duplication or compromise of administrative accounts.

Key Responsibilities of the Super Admin Account:

- **Protocol Code Management**: The Super Admin Account has exclusive authority to upgrade and modify the protocol's smart contracts (usually the program deployer).
- **Configuration Management**: It can create and update configuration accounts, ensuring proper governance of protocol parameters.
- **Fee Collection Authority**: Assigns and manages the `configAuthority`, which collects protocol fees and manages their distribution.
- **Initialize reinvestment account:** Initializes a separate reinvestment account that is responsible for reinvesting earned liquidity providers' fees for reinvestment purposes

Access to the Super Admin Account is tightly controlled. Any transaction involving the creation or modification of configuration accounts must be signed by the Super Admin. The transaction is reverted with an appropriate error message if an unauthorized key is detected.

Technical Implementation:

- The PDA for the Super Admin Account is initialized during the protocol setup phase.
- The Super Admin Account stores only the public key.

2.1.2 Configuration Accounts

Configuration Accounts are essential for managing and storing the protocol's critical parameters. They operate under the authority of the Super Admin Account and serve as the central repository for all adjustable settings.

Key Attributes of Configuration Accounts:

1. **Protocol Fee Management**:
    - Store the `defaultProtocolFeeRate`, representing the percentage of fees allocated to the protocol.
    - Facilitate dynamic updates to fee structures without redeploying smart contracts.
2. **Swap Referral Fee Configuration**:
    - Define and adjust the `defaultSwapReferralRewardFeeRate`, ensuring the definition of the default fee value for the swap referral system.
3. **Protocol Governance**:
    - Hold key administrative parameters, such as `configAuthority`, to streamline management processes.
4. **Security Controls**:
    - Include verification mechanisms to ensure that only authorized accounts can initiate modifications.

Technical Implementation:

- Each Configuration Account is a PDA derived from the Super Admin Account's PDA, default protocol fee rate, and a seed string.
- Example fields in the Configuration Account structure:

- `configAuthority`: Public key of the account managing protocol configurations.
- `defaultprotocolFeeRate`: Default fee rate allocated to the protocol (e.g., in basis points).
- `defaultSwapReferralRewardFeeRate`: Default fee rate allocated to the swap referrer.

2.2 Fee Structure

The protocol implements a robust multi-tiered fee structure designed to optimize trading efficiency while equitably distributing fees between the protocol and liquidity providers (LPs). The fee structure consists of two primary components: **Fee Tiers** and the **Distribution of Fees**.

2.2.1 Fee Tiers and Their Configurations

The Fee Tier system is implemented as a **Program Derived Address (PDA)** account, ensuring unique and deterministic generation based on the protocol's configuration. Each Fee Tier account encapsulates critical parameters required for the efficient operation of liquidity pools:

Fee Tier Data Structure

```
aiDexConfig: Address of the associated AiDex configuration

tickSpacing: Granularity of price levels

defaultFeeRate: Base fee rate in basis points (bps)
```

Key Parameters:

1. **Fee Rate**:
   - Defines the base fee rate applied to all operations within the liquidity pool.
   - Specified in basis points, where 1 basis point equals 0.01%.
2. **Tick Spacing**:
   - Controls the granularity of price levels (ticks) in the liquidity pool.
   - Smaller tick spacing enables finer granularity, which is ideal for stable pairs.
   - Larger tick spacing suits volatile pairs, optimizing memory and processing efficiency.

Flexibility of Fee Tiers:

Fee Tiers allow the protocol to support a diverse range of trading pairs:

- **Stable Pairs** (e.g., USDC/USDT):
  - Low tick spacing and minimal fees to encourage efficient swaps.

- **Volatile Pairs**:
    - Higher tick spacing and elevated fees to compensate for increased risk.

Configuration and Initialization:

- Fee Tier accounts are linked to the protocol configuration through the `aiDexConfig` public key.
- Unique Fee Tier accounts are generated by combining the public key of the AiDexConfig PDA with the specific tick spacing for the pool.
- This ensures that each Fee Tier is uniquely associated with its corresponding liquidity pool, preventing duplication or conflicts.

2.2.2 Distribution of Protocol and Liquidity Provider Fees

The protocol employs a clear and equitable methodology for calculating and distributing fees collected during swap operations. The process is broken down into the following components:

1. Total Swap Fee

The total fee for a swap operation is calculated using the formula:

$$swap\ fee\ = \frac{input\ amount \times fee\ rate}{1{,}000{,}000}$$

- **Input Amount**: The quantity of tokens being exchanged in the swap.
- **Fee Rate**: The fee rate applicable to the liquidity pool, specified in hundredths of a basis point (1 bps = 0.01%).

2. Protocol Fee

A portion of the total swap fee is allocated to the protocol, as determined by the protocol fee rate:

$$protocol\ fee\ = \frac{swap\ fee \times protocol\ fee\ rate}{10{,}000}$$

- **Protocol Fee Rate**: Defined in basis points (e.g., 300 bps for a 3% protocol fee).
- Maximum supported protocol fee rate: **25% of the fee rate (2,500 in basis points)**.

3. Referral Fee

If a swap referral is included, a portion of the protocol fee is allocated as a referral reward:

$$Referral\ Fee\ Amount\ = \frac{Protocol\ Fee\ Amount \times Referral\ Fee\ Rate}{10{,}000}$$

- Maximum supported referral fee rate: **15% of the protocol fee (1,500 in basis points)**.
- The remaining protocol fee after the referral fee is distributed back to the protocol as revenue.

4. Liquidity Provider Fee

The remaining fee, after deducting the protocol fee, is distributed to liquidity providers:

$$LP\ fee\ =\ swap\ fee\ -\ protocol\ fee$$

2.3 Liquidity Pools

Liquidity pools are the cornerstone of the protocol, facilitating token exchanges and managing liquidity. Each pool is carefully structured and initialized with specific parameters to ensure seamless operation and flexibility.

2.3.1 Initialization and Configuration Parameters

The initialization process for liquidity pools involves defining key attributes and ensuring proper configuration for optimized trading and liquidity management.

Essential Parameters for Pool Initialization:

1. **Token Addresses**:
    - Addresses of tokens A and B forming the trading pair.
2. **Fee Tier**:
    - Determines the fee rate and tick spacing for the pool, aligning with the configured Fee Tier account.
3. **Current Price**:
    - Represented as the square root price scaled by 2^{64} (`sqrtPriceX64`).
    - Calculated using the formula: $sqrtPriceX64\ =\ \sqrt{price}\ \times\ 2^{64}$
4. **PDA Address**:
    - A unique address for the liquidity pool, derived programmatically.
5. **Tick Spacing**:
    - Used to validate compatibility with the selected Fee Tier and to use it as a part of PDA.
6. **Sender's Keys**:
    - Public and private keys of the sender initiating the pool creation.

Initialization Mechanism:

- The protocol generates an instruction containing the above parameters, serializes it, adds it to a transaction, signs it, and submits it to the blockchain.
- Once confirmed, the pool state is created and recorded on-chain.

- It is important to note that the pool creation process is split into 2 separate instructions due to stack memory limitations.

2.3.2 Mechanisms for Price Calculation and Liquidity Management

The liquidity pool maintains a comprehensive data structure to manage token exchanges and liquidity efficiently.

Key Components of the Pool State:

1. **Configuration Parameters**:
    - **Ai Dex Config**: The address of the protocol's configuration account.
    - **Tick Spacing**: Granularity of price levels.
    - **Fee Rate**: Base fee rate for the pool, expressed in basis points (e.g., 300 for 0.3%).
    - **Protocol Fee Rate**: Protocol's share of the fee, also in basis points (e.g., 300 for 0.3%).
2. **Liquidity and Price Tracking**:
    - **Liquidity**: Current active liquidity in the pool, initialized at 0.
    - **Sqrt Price (with x64)**: The square root of the current price,.
    - **Tick Current Index**: Index of the active tick, derived from `sqrt_price_x64`.
3. **Token Vaults**:
    - **Token Mint A/B**: Mint addresses for tokens A and B.
    - **Token Vault A/B**: PDA addresses for storing token reserves.
    - Vaults are initialized empty and populated as liquidity is added or swapped.
4. **Fee Accounting**:
    - **Protocol Fee Owed A/B**: Accumulated protocol fees for tokens A and B.
    - **Fee Growth Global A/B**: Total fee growth for tokens A and B across all positions.
5. **Timestamps**:
    - Manage lifecycle events for temporary pools:
        - **Start Timestamp**: When LPs can begin adding liquidity or swaps can commence.
        - **End Timestamp**: When the pool closes for swaps or liquidity withdrawals.
6. **Additional Parameters**:
    - **Is Temporary Pool**: Boolean flag indicating whether the pool is temporary.
    - **Is Oracle Pool**: Boolean flag indicating whether the pool is with oracle.
    - **Last Updated Oracle Timestamp:** The last timestamp at which the price was updated from oracle.
    - **Reward Infos**: Array of reward details for liquidity providers.
    - **Reward Last Updated Timestamp:** The last timeswap at which the reward was updated

---

2.4 Tick Arrays

Tick Arrays are a critical component of the protocol, enabling efficient management of liquidity and price ranges. By organizing and optimizing tick data, the system ensures scalability and minimizes resource consumption.

2.4.1 Role in Liquidity Management

Ticks represent discrete price levels within a liquidity pool, and Tick Arrays are collections of these levels. They serve as the foundation for tracking and managing liquidity, fees, and rewards within specific price ranges.

Tick Array Structure:

1. **Start Tick Index**:
   - Indicates the starting tick index of the array.
   - Calculated as:

$$Start\ Tick\ Index\ =\ Actual\ Index\ -\ (Actual\ Index\ \%\ TICK\_ARRAY\_SIZE)$$

   - Here, **TICK_ARRAY_SIZE** is the number of ticks in the array, typically set to 88.
2. **Ticks**:
   - An array of 88 ticks, each representing a specific price level.
3. **Associated Liquidity Pool**:
   - Each Tick Array is linked to a specific liquidity pool, identified by the `AiDexPool` public key.

Individual Tick Fields:

1. **Initialized**:
   - Boolean value (`true/false`) indicating whether the tick has been initialized.
   - Ticks are initialized the first time liquidity is added at their price level.
2. **Liquidity Net**:
   - Represents the net change in liquidity when the price crosses this tick:
     - **Positive**: Liquidity increases as the price moves upward.
     - **Negative**: Liquidity decreases as the price moves upward.
3. **Liquidity Gross**:
   - The total absolute liquidity associated with the tick.
   - Always greater than or equal to the absolute value of `Liquidity Net`.
4. **Fee Growth Outside (A/B)**:
   - Accumulated fees outside the range defined by this tick.
   - Used for precise fee calculations for positions spanning multiple ticks.
5. **Reward Growths Outside**:
   - Tracks additional reward growth for up to three token types outside the tick range.
   - Facilitates distribution of rewards to liquidity providers.

2.4.2 Memory Optimization and Scalability

Tick Arrays are designed to minimize on-chain memory usage and ensure the protocol's scalability, even under high transaction volumes.

Price Calculation: Each tick in the array corresponds to a price level calculated using the formula:

$$Price_i = 1.0001^i$$

- **i**: Tick index, representing the discrete price level.

Indexing and Tick Spacing: Tick spacing is used to reduce memory consumption and manage liquidity across price ranges effectively:

Initialized Tick Index=Tick Index−(Tick Index%Tick Spacing)\text{Initialized Tick Index} = \text{Tick Index} - (\text{Tick Index} \% \text{Tick Spacing})Initialized Tick Index=Tick Index−(Tick Index%Tick Spacing)

$$Initialized\ Tick\ Index = Tick\ Index - (Tick\ Index \,\%\, Tick\ Spacing)$$

- **Small Tick Spacing**:
  - Suitable for stable pairs (e.g., USDC/USDT).
  - Provides precise price granularity.
- **Large Tick Spacing**:
  - Used for volatile pairs to optimize resource usage and scalability.

2.5 Positions

Positions in the protocol represent records of liquidity provided by individual liquidity providers (LPs). They are crucial for tracking ownership, calculating fees, and managing rewards across defined price ranges.

2.5.1 Lifecycle of a Position

A position is created, managed, and closed through the following lifecycle stages:

1. **Position Creation**:
   - LPs specify the amount of liquidity they wish to provide, along with the price range (`tickLowerIndex` and `tickUpperIndex`).
   - An NFT is minted as proof of ownership, linking the LP to the position.
2. **Liquidity Management**:

    - LPs can adjust liquidity levels by adding or removing assets within the defined price range.
    - Adjustments update the position's state, including liquidity values and fee accounting.
3. **Fee and Reward Accumulation**:
    - Fees and rewards accumulate as trades occur within the position's price range.
    - Updates to `fee_growth_checkpoint_a/b` ensure accurate tracking of accrued amounts.
4. **Position Closure**:
    - LPs withdraw their liquidity, along with any accrued fees and rewards.
    - The associated NFT is burned, finalizing the position.

2.5.2 Structure of a Position

Each position is structured to efficiently manage liquidity, fees, and rewards. Key components include:

1. **Identification and Ownership**:
    - **`aiDexPool`**: Address of the associated liquidity pool.
    - **`positionMint`**: NFT token address serving as proof of ownership.
2. **Liquidity Parameters**:
    - **`liquidity`**: Amount of liquidity provided, stored in a fixed-point format for precision.
    - **`tickLowerIndex` and `tickUpperIndex`**: Define the price range within which the position is active.
3. **Fee Accounting**:
    - **`fee_growth_checkpoint_a/b`**:
        - Records the global fee growth for tokens A and B at the last update to the position.
    - **`fee_owed_a/b`**:
        - Tracks the accumulated fees ready for collection by the position owner.
4. **Reward Mechanisms**:
    - **`reward_infos`**:
        - An array of up to three slots, each representing a different type of reward.
        - Allows for custom reward programs to incentivize liquidity provision.

2.5.3 Fee Accounting and Reward Mechanisms

The protocol incorporates sophisticated mechanisms for calculating and distributing fees and rewards.

1. **Fee Accounting**:
    - Fees are calculated based on the activity within the position's defined price range.
    - The following process is used:
        - **Global Fee Growth**: Tracks cumulative fees generated in the liquidity pool.
        - **Checkpointing**:
            - At each update, `fee_growth_checkpoint_a/b` records the global fee growth values.
            - Results are added to `fee_owed_a/b`.
2. **Reward Distribution**:
    - Rewards are designed to incentivize long-term liquidity provision and align with protocol goals.
    - **Reward Infos**:
        - Stores reward parameters for each position.
        - Tracks growth outside the position's active range to ensure fairness in distribution.

2.6 Oracle Account

The **Oracle Account** is an essential feature for managing price data in liquidity pools initialized as oracle pools. It ensures accurate and automated price updates, enabling efficient swaps and liquidity management.

2.6.1 Oracle Pool Initialization

When initializing an oracle pool, users must specify additional parameters to ensure proper configuration:

1. **Oracle Account**:
    - A PDA linked to the liquidity pool, storing oracle-specific data.
2. **`priceUpdate`**:
    - The Pyth address for the token pair.
3. **`priceFeedId`**:
    - A unique identifier for the Pyth price feed of the token pair.
4. **`maximumAge`**:
    - Defines the maximum allowable age for price updates to be considered valid.

Key Characteristics of Oracle Pools:

- Oracle pools can only be created for **full-range positions**, where:

$$tickSpacing \ > \ 32768$$

- All liquidity is stored in a single tick array, with: $start\ tick\ index = 0$
  - This simplifies calculations and prevents errors in liquidity updates.
- Price information is extracted from the Pyth price feed during initialization, converted to `sqrt_price_x64`, and stored in the pool.

2.6.2 Oracle Account Structure

The **Oracle Account** is implemented as a PDA and includes the following fields:

```
OracleAccount {

    priceFeedId: String,    // Hash ID for the token pair price feed

    maximumAge: u64,        // Maximum age for valid price updates

    mintA: Pubkey,          // Address of token A

    mintB: Pubkey,          // Address of token B

}
```

Key Details:

- **Price Feed ID**:
  - Identifies the token pair's price feed in the Pyth network.
- **Maximum Age**:
  - Ensures only recent price updates are considered for operations.
- **Token Mints**:
  - Define the tokens managed by the oracle pool.

2.6.3 Functionality of Oracle Account

The Oracle Account is responsible for several critical tasks:

1. **Initialization**:
   - During pool creation, the Oracle Account is initialized with the specified `priceFeedId`, `maximumAge`, and token mints.
   - The `initialize` method validates the token mint order to prevent inconsistencies.
2. **Price Updates**:
   - The account uses data from the Pyth price feed or mock price updates to calculate the `sqrtPriceX64` value for the pool.
   - Updates are triggered during liquidity additions, withdrawals, or swaps.
3. **Automated Adjustments**:

    - The Oracle Account ensures that the liquidity pool’s `sqrtPriceX64` and tick index remain accurate by calling the `updateSqrtPrice` method.
4. **Age Validation**:
    - The `maximumAge` parameter ensures that stale price data is not used. Price updates older than the specified limit are rejected.
5. **Customizations**:
    - The maximum age can be adjusted post-initialization using the `changeMaximumAge` method.

2.7 Reinvestments Account

The **Reinvestments Account** is an administrative account responsible for managing reinvestment operations within the protocol. It allows users to automatically reinvest their earned liquidity provider (LP) commissions back into their positions, increasing liquidity seamlessly.

2.7.1 Purpose and Functionality

The Reinvestments Account was designed to:

1. Automate reinvestment of earned LP commissions for users who opt-in during position creation.
2. Minimize user overhead by handling reinvestments through protocol-managed operations.
3. Charge a minimal reinvestment fee to cover the cost of transactions generated during the reinvestment process.

2.7.2 Reinvestments Account Structure

The **Reinvestments Account** is initialized by the **Super Admin** and includes the following fields:

```
AiDexReinvestments {

    reinvestmentsAuthority: // Authority for managing reinvestments

    defaultReinvestmentFeeRate: // Fee rate for reinvestment
operations

}
```

Key Details:

1. **Reinvestments Authority**:
   - A public key identifying the administrative account responsible for managing reinvestments.
   - Can be updated by the Super Admin to delegate authority.
2. **Default Reinvestment Fee Rate**:
   - The standard fee rate charged for reinvestment transactions.

2.7.3 Reinvestments Workflow

The reinvestment process involves several steps, handled automatically by the protocol:

1. **User Opt-In**:
   - During position creation, users can opt to enable reinvestments for their positions.
   - This preference is recorded in the position's state.
2. **Earning LP Commissions**:
   - As trades occur within the position's price range, fees are earned and tracked in the position's `fee_owed_a/b` fields.
3. **Automatic Reinvestment**:
   - When triggered, the reinvestment process:
     - Collects the earned fees.
     - Deducts the **reinvestment fee** based on the `defaultReinvestmentFeeRate`.
     - Converts the net amount into liquidity and reinvests it back into the position.
4. **Administrative Oversight**:
   - The **Reinvestments Authority** manages the execution of reinvestment transactions and ensures compliance with protocol parameters.

Administrative Functions

The **Reinvestments Account** includes several key administrative functions:

1. **Initialization**:
   - The account is initialized by the Super Admin with the `reinvestmentsAuthority` and the `defaultReinvestmentFeeRate`.
2. **Updating Fee Rate**:
   - The `defaultReinvestmentFeeRate` can be updated, ensuring it does not exceed the maximum allowed rate.
3. **Updating Authority**:
   - The `reinvestments_authority` can be changed by the Super Admin to delegate control.

2.8 Swap Referral Account

The **Swap Referral Account** is a mechanism designed to reward referrers (User A) for bringing new users (User B) to the protocol. This account allows a portion of the protocol fee to be shared with the referrer during swaps, incentivizing user engagement and protocol growth.

2.8.1 Purpose and Functionality

The Swap Referral Account enables a referrer (User A) to:

1. Create a referral account that includes their address, referral code, and reward fee rate.
2. Share their referral code with other users (User B) who can apply it during swaps.
3. Automatically receive a portion of the protocol fee as a reward when referred users complete swaps.

Referred users specify the referrer's account during swaps, allowing the protocol to allocate a percentage of the protocol fee to the referrer. Funds are initially deposited into an **Associated Token Account (ATA)** owned by the Swap Referral Account. Referrers can withdraw their accumulated rewards at any time.

2.8.2 Swap Referral Account Structure

The **Swap Referral Account** includes the following fields:

```
SwapReferral {

    referrerAddress: Referrer's public key

    referralRewardFeeRate: Reward fee rate for the referrer

    referralCode: Unique referral code

    referralBump: Bump seed for PDA generation

}
```

Key Details:

1. **Referrer Address**:
   - Identifies the owner of the referral account.
2. **Referral Reward Fee Rate**:
   - Specifies the percentage of the protocol fee allocated to the referrer.

    - Maximum allowed rate: **15% of the protocol fee** (`MAX_REFERRAL_REWARD_FEE_RATE`).
3. **Referral Code**:
    - A unique identifier for the referral program, allowing referred users to associate swaps with the referrer.
4. **Referral Bump**:
    - Ensures deterministic PDA generation for the Swap Referral Account.

2.8.3 Referral Workflow

1. **Creating a Referral Account**:
    - Referrers (User A) initialize a **Swap Referral Account** with their public key, a unique referral code, and a `referral_bump`.
    - The `initializeSwapReferral` method sets up the account and initializes the referral fee rate to 0.
2. **Specifying a Referral During Swap**:
    - Referred users (User B) include the Swap Referral Account in their swap transaction, identifying it by the referral code.
    - The protocol allocates a percentage of the protocol fee to the referrer based on the account’s `referralRewardFeeRate`.
3. **Fee Allocation**:
    - The protocol calculates the referrer’s reward.
    - The fee is deposited into an ATA owned by the Swap Referral Account.
4. **Reward Withdrawal**:
    - Referrers can withdraw accumulated rewards from the ATA to their personal ATA by initiating a withdrawal transaction.

Administrative Functions

The Swap Referral Account includes the following administrative methods:

1. **Initialize Swap Referral**:
    - Sets up the account with the referrer’s address, referral code, and bump seed.
2. **Update Reward Fee Rate**:
    - Updates the referral reward fee rate, ensuring it does not exceed the maximum allowed rate.
3. **Seed Access**:
    - Provides access to the account’s seeds for PDA generation.

2.9 Trade Batch Positions

The **Trade Batch Positions** feature allows users to manage multiple liquidity positions (up to 256) under a single Non-Fungible Token (NFT). This capability simplifies position management, enhances flexibility, and reduces on-chain storage overhead.

2.9.1 Purpose and Functionality

Trade Batch Positions provide users with:

1. **Simplified Management**:
   - Aggregate multiple positions under one NFT, reducing the complexity of individual position tracking.
2. **Efficiency**:
   - Optimize on-chain resources by grouping positions within a single data structure.
3. **Flexibility**:
   - Enable diverse strategies by allowing users to define distinct parameters for each position within the batch.

2.9.2 Structure of Trade Batch Positions

Each batch is represented as a single entity tied to an NFT, with the following structure:

1. **Batch NFT**:
   - Represents ownership of the batch and all associated positions.
   - Tied to the batch via a unique `positionMint`.
2. **Position Parameters**:
   - Each position in the batch includes:
     - **Liquidity**:
       - Amount of liquidity allocated to the position.
     - **Price Range**:
       - Defined by `tickLowerIndex` and `tickUpperIndex` for each position.
     - **Fee Accumulation**:
       - Tracks fees earned (`feeOwedA/B`) for each position.
     - **Rewards**:
       - Supports up to three reward types, tracked individually for each position.
3. **Batch Size**:
   - The batch can contain up to 256 positions.
   - Each position is indexed within the batch for efficient access and updates.

---

3. Workflows

3.1 Admin Workflows

The administrative workflows provide the necessary tools for configuring and managing the protocol to ensure its smooth operation and adaptability to market needs. These tasks are restricted to specific accounts with elevated privileges, such as the **Super Admin** or configuration accounts.

3.1.1 Super Admin Actions

1. **Initializing the DEX Configuration**:
   - **Purpose**:
     - Establish the initial configuration of the protocol, including creating the global configuration account, setting default parameters, and initializing key PDAs.
   - **Steps**:
     - Deploy the smart contracts.
     - Set up global configuration values (e.g., protocol-wide default fees, reward rates, reinvestment parameters).
     - Define the Super Admin's authority, ensuring secure access to critical admin functionalities.
2. **Managing Super Admin Authority**:
   - **Purpose**:
     - Enable the delegation or transfer of administrative privileges when necessary.
   - **Steps**:
     - Update the public key for the Super Admin account to reflect a new authority.
     - Ensure that all subsequent admin actions are authorized by the updated Super Admin key.
3. **Upgrading Protocol Components**:
   - **Purpose**:
     - Deploy new versions of smart contracts to add features or fix vulnerabilities.
   - **Steps**:
     - Use the Super Admin account to initiate and approve the upgrade process.
     - Verify that existing configurations and state are compatible with the new version.

3.1.2 Configuration Management

1. **Configuring Protocol-Wide Parameters**:

  - **Purpose**:
    - Adjust global settings that affect all pools and positions.
  - **Steps**:
    - Access the global configuration account via the Super Admin.
    - Update parameters such as the maximum allowable price update age, default fee rates, and reinvestment settings.
2. **Updating Default Reinvestment Fee Rates**:
   - **Purpose**:
     - Modify the protocol-wide default fee rate for reinvestment operations to ensure competitiveness and cover transaction costs.
   - **Steps**:
     - Call the setter function on the **Reinvestments Account** with the new default fee rate.
3. **Adjusting Maximum Price Update Age for Oracle Pools**:
   - **Purpose**:
     - Define the maximum time interval for valid price updates in oracle pools to maintain accurate pricing.
   - **Steps**:
     - Access the specific **Oracle Account** for the pool.
     - Update the `maximumAge` parameter to reflect the desired value.
     - Verify that the updated age parameter aligns with the requirements of the connected price feeds.
4. **Updating Default Fee Rates for Liquidity Pools**:
   - **Purpose**:
     - Modify the default LP fee rate for all pools linked to a specific Fee Tier.
   - **Steps**:
     - Access the target **Fee Tier** account.
     - Update the `defaultFeeRate` parameter via the admin setters.
     - Verify that the new rate is consistent with the protocol's objectives.

3.1.3 Incentive Management

1. **Adding Rewards to Liquidity Pools**:
   - **Purpose**:
     - Incentivize liquidity providers by allocating additional rewards to specific pools.
   - **Steps**:
     - Access the reward configuration for the target pool.
     - Specify the reward type (e.g., token) and the total reward allocation.
     - Define the reward distribution schedule and parameters.
2. **Managing Reward Parameters Across Pools**:
   - **Purpose**:
     - Adjust reward distribution rules to align with market conditions or strategic goals.
   - **Steps**:
     - Update the reward growth settings for each tick in the pool.

        - Modify the reward emission rates to optimize liquidity distribution.
        - Remove or reallocate rewards as necessary.

3.2 User Workflows

3.2.1 Pool and Fee Tier Initialization

Users play an active role in setting up the necessary components for liquidity provisioning and trading on the protocol. This section describes the workflows for initializing Fee Tiers and Liquidity Pools.

Initializing a Fee Tier

Purpose:
A **Fee Tier** defines the fee rate and tick spacing for liquidity pools. Users create Fee Tiers to optimize liquidity and trading for specific token pairs based on their volatility and use case.

Workflow:

1. **Input Parameters**:
    - **`aiDexConfig`**:
        - Public key of the protocol configuration account.
    - **`tickSpacing`**:
        - Granularity of the price levels for the liquidity pool.
        - Example:
            - Stable pairs (e.g., USDC/USDT): Small tick spacing (e.g., 1).
            - Volatile pairs: Larger tick spacing (e.g., 256).
    - **`defaultFeeRate`**:
        - Fee rate in basis points for trades executed in the pools under this Fee Tier.
2. **Steps**:
    - Call the Fee Tier initialization instruction in the protocol smart contract.
    - Provide the input parameters (`aiDexConfig`, `tickSpacing`, `defaultFeeRate`).
    - The protocol generates a **Program Derived Address (PDA)** for the Fee Tier based on the provided parameters.
3. **Outcome**:
    - A new Fee Tier account is created with the specified configuration.
    - This Fee Tier can now be referenced during the initialization of liquidity pools.

Initializing a Liquidity Pool

Purpose:
A **Liquidity Pool** facilitates token swaps and manages liquidity for a specific token pair. Users initialize a pool by specifying the token pair, Fee Tier, and other parameters.

Workflow:

1. **Input Parameters**:
   - **Token Pair**:
     - Addresses of **token A** and **token B** for the trading pair.
   - **Fee Tier**:
     - The previously created Fee Tier, specifying `tick_spacing` and `default_fee_rate`.
   - **`sqrtPriceX64`**:
     - Initial price of the token pair.
   - **`tickSpacing`**:
     - Validated against the selected Fee Tier to ensure compatibility.
   - **Sender's Keys**:
     - Public and private keys of the user initializing the pool.
2. **Steps**:
   - Specify the parameters in the liquidity pool initialization transaction.
   - Generate the **PDA Address** for the liquidity pool.
   - The protocol:
     - Validates the input parameters.
     - Creates the liquidity pool account.
     - Initializes the pool state.
3. **Outcome**:
   - A new Liquidity Pool is created and associated with the specified Fee Tier.
   - The pool is now ready for users to open positions and add liquidity.

Example Use Cases

1. **Stable Token Pair**:
   - Initialize a Fee Tier with low tick spacing and minimal fees for stablecoin pairs.
   - Create a liquidity pool under this Fee Tier for efficient trading between USDC and USDT.
2. **Volatile Token Pair**:
   - Use a Fee Tier with higher tick spacing to accommodate volatile token pairs.
   - Create a liquidity pool under this Fee Tier for trading between SOL and a volatile token.

3.2.2 Position Management

The position management workflows allow users to interact with liquidity pools by opening, modifying, and closing positions. These workflows ensure flexibility and efficiency in managing liquidity.

Opening Positions on a Pool

Purpose:
To allow users to define a liquidity position in a specific price range on a pool.

Workflow:

1. **Input Parameters**:
   - **Pool Address**:
     - The PDA of the liquidity pool.
   - **Price Range**:
     - Defined by `tickLowerIndex` and `tickUpperIndex`, representing the bounds of the position.
   - **Liquidity Amount**:
     - The fixed-point amount of liquidity to provide.
   - **Sender's Keys**:
     - Public and private keys of the user opening the position.
2. **Steps**:
   - Call the smart contract function for opening a position.
   - Specify the pool address, price range, and liquidity amount.
   - The protocol:
     - Validates the price range.
     - Allocates liquidity to the position.
     - Mints an NFT to represent ownership of the position.
3. **Outcome**:
   - A new position is created, linked to the user's NFT.
   - The position is initialized with the specified price range and liquidity.

Adding Liquidity to a Position

Purpose:
To increase the amount of liquidity in an existing position.

Workflow:

1. **Input Parameters**:
   - **Position data**:
     - Position token address, PDA address position mint address.
   - **Liquidity Amount**:
     - The additional liquidity to add.
   - **Pool Address**:

        - The PDA of the associated liquidity pool.
2. **Steps**:
    - Call the increase liquidity function, specifying needed data.
    - The protocol:
        - Validates the position and pool.
        - Adjusts the `Liquidity Net` and `Liquidity Gross` values for the associated ticks.
        - Updates the position’s liquidity amount.
3. **Outcome**:
    - The liquidity in the position is increased.
    - Fees and rewards continue to accumulate based on the updated liquidity.

Withdrawing Liquidity from a Position

Purpose:
To remove a portion of the liquidity from an existing position.

Workflow:

1. **Input Parameters**:
    - **Position Data**:
        - Position token address, PDA address position mint address.
    - **Liquidity Amount**:
        - The amount of liquidity to withdraw.
    - **Pool Address**:
        - The PDA of the associated liquidity pool.
2. **Steps**:
    - Call the decrease liquidity function, specifying the position NFT and liquidity amount.
    - The protocol:
        - Validates the position and pool.
        - Adjusts the `Liquidity Net` and `Liquidity Gross` values for the associated ticks.
        - Reduces the position’s liquidity by the specified amount.
3. **Outcome**:
    - The withdrawn liquidity is returned to the user.
    - The position remains active with the reduced liquidity.

Closing Positions on a Pool

Purpose:
To fully withdraw liquidity and close the position. Make sure that the liquidity should be equal to zero in the position to close it.

Workflow:

1. **Input Parameters**:
   - **Position Data**:
     - Position token address, PDA address position mint address.
2. **Steps**:
   - Call the close position function, specifying the position NFT.
   - The protocol:
     - Validates the position and pool.
     - Removes all remaining liquidity from the position.
     - Burns the NFT representing the position.
3. **Outcome**:
   - The position is closed.
   - The NFT representing the position is burned.

3.2.3 Earnings and Incentives

The **Earnings and Incentives** workflows allow users to collect their earned commissions and rewards from liquidity positions. These workflows ensure that users receive accurate and updated distributions based on their liquidity contributions and participation in reward programs.

Collecting Earned Commissions

Purpose:
To collect the accumulated trading fees (commissions) from a user's liquidity position.

Workflow:

1. **Prerequisite: Updating Fees**:
   - Before collecting fees, the user must call the `updateFeesAndRewards` instruction to ensure the position's fee information is up-to-date.
   - This updates the `feeOwedA` and `feeOwedB` for the position.
2. **Steps**:
   - Call the `updateFeesAndRewards` function with the needed inputs:
   - The protocol calculates the latest accrued fees by comparing the global fee growth against the position's fee growth checkpoint.
3. **Collecting Fees**:
   - After updating, call the `collectFees` function with the needed inputs:
   - The protocol:
     - Validates the position and pool.
     - Transfers the accumulated fees (`feeOwedA` and `feeOwedB`) to the user's provided associated token accounts (ATAs).
4. **Outcome**:

- The user receives the earned fees in their ATAs.
- The position's `feeOwedA` and `feeOwedB` fields are reset to zero.

Participating in Reward Programs

Purpose:
To collect additional rewards distributed as part of liquidity pool incentive programs.

Workflow:

1. **Prerequisite: Updating Rewards**:
    - Similar to fees, users must call the `updateFeesAndRewards` instruction to ensure reward information is up-to-date. FYI, you can call it all together in one transaction
    - This updates the `rewardInfos` field for the position.
2. **Steps**:
    - Call the `updateFeesAndRewards` function with the needed inputs:
    - The protocol updates reward growth fields to calculate the latest rewards.
3. **Collecting Rewards**:
    - After updating, call the `collectRewards` function with the needed inputs:
    - The protocol:
        - Validates the position and pool.
        - Transfers the collected rewards to the user's ATA for the specified reward type.
4. **Outcome**:
    - The user receives the rewards in their ATA.
    - The relevant reward fields in the position are reset or adjusted as necessary.

Example Use Case

1. **Scenario**:
    - A user has provided liquidity to a pool and wants to collect their earnings.
2. **Steps**:
    - Call `updateFeesAndRewards` to calculate accrued fees and rewards.
    - Call `collectFees` to transfer trading fees to their ATAs.
    - Call `collectRewards` for additional incentive tokens.

3.2.4 Referral Programs

Purpose:
To incentivize user engagement and protocol adoption by enabling referrers (User A) to earn rewards from referred swaps executed by other users (User B). The rewards are calculated

from the protocol fee and stored in a referral account, allowing referrers to withdraw them later.

Workflow: Creating and Using a Referral Account

1. Creating a Referral Account (User A):

- **Objective**: User A creates a **Swap Referral Account** to earn rewards from referred swaps.
- **Steps**:
    1. User A initializes a referral account by calling the `initializeSwapReferral` instruction with needed params:
    2. The protocol:
        - Creates a **Swap Referral Account** PDA linked to User A.
        - Initializes the `referralRewardFeeRate` to zero (the default value would be used each time, but we can change it from 0 to not use the default one).
    3. User A receives the referral code to share with others.

---

2. Using a Referral During a Swap (User B):

- **Objective**: User B uses User A's referral code during a swap.
- **Steps**:
    1. User B specifies the referral account in the swap transaction.
    2. The protocol:
        - Executes the swap and calculates the **protocol fee**.
        - Calculates the **referral fee**.
        - Deducts the referral fee from the protocol fee.
        - Credits the referral fee to User A's referral account.
    3. The remaining protocol fee is retained by the protocol.

---

3. Storing Referral Rewards:

- **Objective**: Store the calculated referral fee in the **Swap Referral Account** linked to User A.
- **Steps**:
    1. The referral fee is added to the **ATA** (Associated Token Account) owned by the Swap Referral Account PDA.
    2. The account maintains a running total of earned rewards, accessible by User A.

---

4. Withdrawing Referral Rewards (User A):

- **Objective**: User A withdraws accumulated referral rewards to their personal ATA.
- **Steps**:
    1. User A calls the `collectReferralFee` function with needed params:
    2. The protocol:
        - Validates the withdrawal request to ensure User A is the referrer.
        - Transfers the accumulated rewards from the referral account's ATA to User A's ATA.
        - Resets the reward balance in the referral account to zero.

Example Interaction

1. **Referral Account Creation**:
    - User A creates a referral account with the referral code **"SWAP123"** and shares it with User B.
2. **Swap with Referral**:
    - User B performs a swap for a token pair and specifies **"SWAP123"** as the referral code.
    - The protocol:
        - Executes the swap with a protocol fee of **1% (as an example)**.
        - Allocates **10% (as an example)** of the protocol fee (based on the referral reward fee rate) to User A's referral account.
3. **Reward Withdrawal**:
    - User A periodically withdraws accumulated rewards to their personal ATA, ensuring their earned referral rewards are accessible.

3.2.5 Token Swaps

This section explains the workflows for executing swaps and two-hop swaps for token exchanges in detail. These workflows enable efficient and secure token trading within the protocol.

Executing Swaps

Purpose:
To exchange one token for another within a single liquidity pool.

Workflow:

1. **Input Parameters**:
    - **Source Token**:
        - The token the user provides for the swap.

    - **Destination Token**:
        - The token the user wants to receive.
    - **Amount**:
        - The amount of the source token for the swap.
    - **Other Amount Threshold**:
        - A minimum acceptable output or maximum acceptable input to protect against slippage.
    - **Sqrt Price Limit**:
        - A boundary for the pool price during the swap.
    - **Amount Specified as Input**:
        - Indicates whether the provided amount is the input or output.
    - **Swap Direction**:
        - Defines the direction of the swap (e.g., from token A to token B).
2. **Steps**:
    - **Validation**:
        - Verify the pool configuration, including token accounts and vaults.
        - Ensure that the user's token accounts match the liquidity pool.
    - **Oracle Update (If Applicable)**:
        - Update the pool price using the oracle account if the pool is configured as an oracle pool.
    - **Fee Calculation**:
        - Compute the protocol fee based on the swap amount and rate.
        - If a referral account is used, calculate the referral fee and allocate it to the referrer's account.
    - **Swap Execution**:
        - Adjust the pool's liquidity and tick indices based on the swap direction and input amount.
        - Update the pool state, including global fee growth and liquidity metrics.
    - **Threshold Validation**:
        - Confirm the output (or input) amount satisfies the specified threshold to ensure slippage is within acceptable limits.
3. **Outcome**:
    - The user receives the destination token in their associated token account.
    - Protocol and referral fees are calculated and distributed accordingly.

Executing Two-Hop Swaps for Token Exchanges

Purpose:
To enable token exchanges for pairs without direct liquidity pools by routing through an intermediate token and two liquidity pools.

Workflow:

1. **Input Parameters**:
    - **Source Token**:

            - The token the user provides for the swap.
    - **Intermediate Token**:
        - The token used as a bridge between the source and destination tokens.
    - **Destination Token**:
        - The token the user wants to receive.
    - **Amount**:
        - The amount of the source token for the swap.
    - **Other Amount Threshold**:
        - A minimum acceptable output or maximum acceptable input to protect against slippage.
    - **Amount Specified as Input**:
        - Indicates whether the provided amount is the input or output.
    - **Sqrt Price Limits**:
        - Defines the price boundaries for each pool in the two-hop swap.
    - **Swap Directions**:
        - Specifies the direction of swaps for both pools.
2. **Steps**:
    - **Validation**:
        - Verify all involved tokens and pools, ensuring alignment of intermediate tokens between the two pools.
        - Confirm that the pools are distinct and correctly configured.
    - **Oracle Updates (If Applicable)**:
        - Update prices for both pools using their respective oracle accounts.
    - **First Swap Execution**:
        - Perform the first swap, exchanging the source token for the intermediate token in the first pool.
        - Compute the output amount of the intermediate token and update the state of the first pool.
    - **Second Swap Execution**:
        - Use the intermediate token from the first swap as input for the second swap, exchanging it for the destination token in the second pool.
        - Update the state of the second pool accordingly.
    - **Fee and Referral Distribution**:
        - Compute and distribute protocol and referral fees for both swaps.
    - **Threshold Validation**:
        - Confirm the final output (or input) amount meets the user-defined threshold.
3. **Outcome**:
    - The user receives the destination token in their associated token account.
    - The protocol and referral fees are allocated appropriately, and the states of both pools are updated.

Key Features of the Swaps Workflow

1. **Slippage Protection**:

    - Users can set thresholds to ensure they receive a fair amount of tokens, protecting against excessive price impact.
2. **Oracle Integration**:
    - Oracle-enabled pools dynamically update prices to ensure accuracy during swaps.
3. **Referral Incentives**:
    - Referral accounts enable fee sharing for swaps, encouraging user-driven growth.
4. **Seamless Two-Hop Swaps**:
    - Efficient handling of multi-pool swaps expands trading possibilities for token pairs without direct pools.
