Uniswap v3: Technical Report

Protocol: Uniswap v3

Solidity Version (Core): `=0.7.6`

Solidity Version (Periphery): `=0.7.6`

Scope: v3-core (`UniswapV3Factory`, `UniswapV3Pool`) + v3-periphery (`NonfungiblePositionManager`, `SwapRouter`, `Quoter`, `UniswapV3Library`)

Analyzed by: rubencrxz

Table of Contents

1. Protocol Overview
2. Architecture & Contract Hierarchy
3. Data Structures & Storage Layout
4. Concentrated Liquidity Model
5. Tick System & Price Mapping
6. Tick Bitmap: Efficient Range Navigation
7. Fixed-Point Math & sqrtPriceX96
8. Factory: Pool Deployment
9. Pool: Swap Mechanics
10. Pool: Liquidity Provisioning (Mint & Burn)
11. Fee Accounting System
12. Price Oracle (Observations Array)
13. Flash Loans
14. Periphery: NonfungiblePositionManager
15. Periphery: SwapRouter & Quoter
16. Protocol Constants & Bounds
17. Notable Design Decisions

1. Protocol Overview

Uniswap v3 is a **concentrated liquidity automated market maker (CLAMM)** — the most significant architectural departure from the constant-product AMM introduced in v1/v2. Rather than deploying liquidity uniformly across the entire price range `(0, ∞)`, v3 allows liquidity providers (LPs) to concentrate capital within user-specified price ranges `[tickLower, tickUpper]`. This design makes capital dramatically more efficient: the whitepaper demonstrates that a v3 position providing the same effective depth as a v2 position requires only a fraction of the capital.

The core protocol is divided into two layers:

- **Core** (`v3-core`): A minimal, fund-holding primitive. Each pool manages a single token pair at a fixed fee tier. Implements the CLAMM invariant, manages the tick system, accumulates fee growth per unit of liquidity, stores oracle observations, and handles swap/mint/burn through low-level callbacks.
- **Periphery** (`v3-periphery`): User-facing layer. The `NonfungiblePositionManager` wraps individual positions as ERC-721 NFTs, enabling composability. `SwapRouter` handles path-encoded multi-hop swaps. `Quoter` provides off-chain quote simulation.

Key improvements over Uniswap v2:

Feature	v2	v3
Liquidity distribution	Uniform `(0, ∞)`	Concentrated `[tickLower, tickUpper]`
Capital efficiency	Low (uniform deployment)	Up to 4000× higher at current price
Position representation	Fungible LP tokens (ERC-20)	Non-fungible positions (ERC-721 via periphery)
Fee tiers	Single (0.30%)	Multiple: 0.01%, 0.05%, 0.30%, 1.00%
Oracle	UQ112x112 TWAP accumulators	Geometric TWAP with 65,535-entry ring buffer
Price representation	`uint112` reserves	`sqrtPriceX96` (Q64.96 fixed-point)
Flash borrowing	Flash swaps (within swap callback)	Dedicated `flash()` function
Tick system	None	Discrete ticks, bitmaps, per-tick fee tracking
Solidity version	0.5.16 / 0.6.6	0.7.6

The contracts are non-upgradeable. The only privileged operation is the ability to enable new fee tiers via `Factory.enableFeeAmount()` and to set a protocol fee per pool via `Pool.setFeeProtocol()`.

---

2. Architecture & Contract Hierarchy

```
v3-core
├── UniswapV3Factory
│   ├── createPool(tokenA, tokenB, fee) → UniswapV3Pool   [CREATE2]
│   ├── getPool[token0][token1][fee] → address
│   ├── feeAmountTickSpacing[fee] → tickSpacing           [fee tier registry]
│   ├── enableFeeAmount(fee, tickSpacing)                  [owner only]
│   └── owner                                              [can set protocol fees]
│
└── UniswapV3Pool
    ├── initialize(sqrtPriceX96)
    ├── swap(recipient, zeroForOne, amountSpecified, sqrtPriceLimitX96, data)
    ├── mint(recipient, tickLower, tickUpper, amount, data) → (amount0, amount1)
    ├── burn(tickLower, tickUpper, amount) → (amount0, amount1)
    ├── collect(recipient, tickLower, tickUpper, amount0Requested, amount1Requested)
    ├── flash(recipient, amount0, amount1, data)
    ├── increaseObservationCardinalityNext(observationCardinalityNext)
    ├── setFeeProtocol(feeProtocol0, feeProtocol1)
    ├── collectProtocol(recipient, amount0Requested, amount1Requested)
    │
    ├── slot0                [sqrtPriceX96, tick, observationIndex, ...]
    ├── feeGrowthGlobal0X128  [Q128.128 fee accumulator for token0]
    ├── feeGrowthGlobal1X128  [Q128.128 fee accumulator for token1]
    ├── protocolFees          [accumulated protocol fees]
    ├── liquidity             [active virtual liquidity]
    ├── ticks[tick]           [per-tick state]
    ├── tickBitmap[word]      [packed tick initialization bitmap]
    ├── positions[key]        [per-position state]
    └── observations[index]   [ring buffer for TWAP oracle]

v3-periphery
├── NonfungiblePositionManager   (ERC-721)
│   ├── mint(MintParams) → (tokenId, liquidity, amount0, amount1)
```

```
    │   ├── increaseLiquidity(IncreaseLiquidityParams) → (liquidity, amount0, amount1)
    │   ├── decreaseLiquidity(DecreaseLiquidityParams) → (amount0, amount1)
    │   ├── collect(CollectParams) → (amount0, amount1)
    │   └── positions[tokenId] → Position
    │
    ├── SwapRouter
    │   ├── exactInputSingle(ExactInputSingleParams) → amountOut
    │   ├── exactInput(ExactInputParams) → amountOut          [multi-hop]
    │   ├── exactOutputSingle(ExactOutputSingleParams) → amountIn
    │   └── exactOutput(ExactOutputParams) → amountIn         [multi-hop]
    │
    └── Quoter
        ├── quoteExactInputSingle(...)
        ├── quoteExactInput(...)
        ├── quoteExactOutputSingle(...)
        └── quoteExactOutput(...)
```

All pools are deployed via `CREATE2` with salt = `keccak256(abi.encode(token0, token1, fee))`, producing deterministic, counterfactual addresses before deployment.

---

3. Data Structures & Storage Layout

3.1 `slot0` — Packed Pool State (1 storage slot)

The most frequently accessed pool state is packed into a single 256-bit storage slot to minimize gas costs. This struct is read on every swap:

```
// UniswapV3Pool.sol
struct Slot0 {
    // the current price
    uint160 sqrtPriceX96;       // bits [0..159]   — Q64.96 sqrt(price)
    // the current tick
    int24   tick;               // bits [160..183]  — current tick index
    // the most-recently updated index of the observations array
    uint16  observationIndex;   // bits [184..199]  — ring buffer pointer
    // the current maximum number of observations that are being stored
    uint16  observationCardinality;      // bits [200..215]
    // the next maximum number of observations to store, triggered in initialize/swap
    uint16  observationCardinalityNext;  // bits [216..231]
    // the current protocol fee as a percentage of the swap fee taken on withdrawal
    // represented as an integer denominator (1/x) %
    uint8   feeProtocol;        // bits [232..239]  — upper 4 bits: fee1, lower 4 bits: fee0
    // whether the pool is locked
    bool    unlocked;           // bit  [240]       — reentrancy guard
}
```

Bit-level layout of `slot0`:

Bits	Field	Size	Notes
0–159	`sqrtPriceX96`	160 bits	Q64.96 fixed-point
160–183	`tick`	24 bits	Signed, range `[MIN_TICK, MAX_TICK]`
184–199	`observationIndex`	16 bits	Current ring buffer write head
200–215	`observationCardinality`	16 bits	Active observations count
216–231	`observationCardinalityNext`	16 bits	Target cardinality after next write
232–239	`feeProtocol`	8 bits	High nibble = fee1, low nibble = fee0
240	`unlocked`	1 bit	Reentrancy mutex
241–255	(padding)	15 bits	Unused

3.2 `Tick.Info` — Per-Tick State

Each initialized tick stores:

```
// libraries/Tick.sol
struct Info {
    // the total position liquidity that references this tick
    uint128 liquidityGross;
    // amount of net liquidity added (subtracted) when tick is crossed going left (right)
    int128  liquidityNet;
    // fee growth per unit of liquidity on the _other_ side of this tick (relative to current)
    uint256 feeGrowthOutside0X128;
    uint256 feeGrowthOutside1X128;
    // the cumulative tick value on the other side of the tick
    int56   tickCumulativeOutside;
    // the seconds per unit of liquidity on the _other_ side of this tick
    uint160 secondsPerLiquidityOutsideX128;
    // the seconds spent on the other side of the tick
    uint32  secondsOutside;
    // true iff the tick is initialized (has been referenced by at least one position)
    bool    initialized;
}
```

The `liquidityNet` field encodes the change in active liquidity when the tick is crossed:

- **Entering a range** (crossing `tickLower` left→right): `+liquidityNet`
- **Exiting a range** (crossing `tickUpper` left→right): `-liquidityNet`
- Symmetrically reversed when traversing right→left

This design allows the pool to update `liquidity` with a single integer addition/subtraction at each tick crossing without iterating over all positions.

3.3 `Position.Info` — Per-Position State

```
// libraries/Position.sol
struct Info {
    // the amount of liquidity owned by this position
    uint128 liquidity;
    // fee growth per unit of liquidity as of the last update to liquidity or fees owed
    uint256 feeGrowthInside0LastX128;
    uint256 feeGrowthInside1LastX128;
    // the fees owed to the position owner in token0/token1
    uint128 tokensOwed0;
    uint128 tokensOwed1;
}
```

Position key derivation:

```
// UniswapV3Pool.sol
bytes32 key = keccak256(abi.encodePacked(owner, tickLower, tickUpper));
```

Each position is uniquely identified by `(owner, tickLower, tickUpper)`. Since the core pool tracks positions by the **caller's address**, the NonfungiblePositionManager is always the owner from the pool's perspective, and it maintains its own `tokenId → position` mapping.

3.4 `Oracle.Observation` — Ring Buffer Entry

```
// libraries/Oracle.sol
struct Observation {
    // the block timestamp of the observation
    uint32  blockTimestamp;
    // the tick accumulator: sum of (tick * delta_t) over all observations
    int56   tickCumulative;
    // the seconds per liquidity accumulator: sum of (1/liquidity * delta_t)
    uint160 secondsPerLiquidityCumulativeX128;
    // whether or not the observation is initialized
    bool    initialized;
}
```

The ring buffer can hold up to 65,535 observations and is expanded by calling `increaseObservationCardinalityNext()`. By default, a newly initialized pool has cardinality = 1.

3.5 Global Pool Storage Variables

```
// State variables in UniswapV3Pool
Slot0 public override slot0;

uint256 public override feeGrowthGlobal0X128;  // Q128.128 accumulated fee per liquidity (token0)
uint256 public override feeGrowthGlobal1X128;  // Q128.128 accumulated fee per liquidity (token1)

ProtocolFees public override protocolFees;     // {uint128 token0, uint128 token1}

uint128 public override liquidity;             // virtual liquidity at current price

mapping(int24 => Tick.Info)    public override ticks;
mapping(int16 => uint256)      public override tickBitmap;
mapping(bytes32 => Position.Info) public override positions;
Oracle.Observation[65535] public override observations;
```

---

4. Concentrated Liquidity Model

4.1 The Virtual Reserve Abstraction

In Uniswap v2, a pool holds real reserves `(x, y)` and enforces `x · y = k`. In v3, the pool only holds **the portion of reserves needed to cover the active liquidity at the current price**. Liquidity outside the current range remains dormant and generates no fees.

The invariant within a tick range is the same CPAMM formula, but parameterized differently. For a position with liquidity `L` spanning `[p_a, p_b]`, the virtual reserves are:

```
x_virtual = L · (√p_b - √p) / (√p · √p_b)
y_virtual = L · (√p - √p_a)
```

Where `p` is the current price (token1 per token0), `p_a = 1.0001^tickLower`, `p_b = 1.0001^tickUpper`.

The AMM still enforces `x_virtual · y_virtual = L²` within each tick range, but `L` changes when the price crosses a tick boundary.

4.2 Capital Efficiency

For a position providing depth at a price `p`, with range `[p_a, p_b]`, the capital efficiency relative to v2 is:

```
efficiency = 1 / (1 - √(p_a/p_b))
```

A ±1% range around the current price yields ~100× efficiency. A ±0.1% range yields ~1000×. Full-range positions (`tickLower = MIN_TICK, tickUpper = MAX_TICK`) are equivalent to v2.

4.3 Price-to-Amount Formulas

Given a swap that moves the price from `√P_a` to `√P_b` within a tick range:

```
Δtoken0 = L · (1/√P_a - 1/√P_b)
Δtoken1 = L · (√P_b - √P_a)
```

These formulas are implemented precisely in `SqrtPriceMath.sol`:

```
// SqrtPriceMath.sol — amount of token0 for a price move
function getAmount0Delta(
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint128 liquidity,
    bool    roundUp
) internal pure returns (uint256 amount0) {
    if (sqrtRatioAX96 > sqrtRatioBX96)
        (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    uint256 numerator1 = uint256(liquidity) << FixedPoint96.RESOLUTION;
    uint256 numerator2 = sqrtRatioBX96 - sqrtRatioAX96;

    require(sqrtRatioAX96 > 0);

    return roundUp
        ? UnsafeMath.divRoundingUp(
            FullMath.mulDivRoundingUp(numerator1, numerator2, sqrtRatioBX96),
            sqrtRatioAX96
          )
        : FullMath.mulDiv(numerator1, numerator2, sqrtRatioBX96) / sqrtRatioAX96;
}

// amount of token1 for a price move
function getAmount1Delta(
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint128 liquidity,
    bool    roundUp
) internal pure returns (uint256 amount1) {
    if (sqrtRatioAX96 > sqrtRatioBX96)
        (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    return roundUp
        ? FullMath.mulDivRoundingUp(liquidity, sqrtRatioBX96 - sqrtRatioAX96, FixedPoint96.Q96)
        : FullMath.mulDiv(liquidity, sqrtRatioBX96 - sqrtRatioAX96, FixedPoint96.Q96);
}
```

---

5. Tick System & Price Mapping

5.1 Tick Definition

A **tick** is a signed 24-bit integer `i ∈ [MIN_TICK, MAX_TICK]` that maps to a specific price point:

```
price(i) = 1.0001^i
```

This exponential spacing means each tick represents a price change of exactly **0.01%** (1 basis point). The mapping between ticks and `sqrtPrice` values is:

```
sqrtPrice(i) = sqrt(1.0001^i) = 1.0001^(i/2)
sqrtPriceX96(i) = sqrtPrice(i) · 2^96
```

Constants:

```
// TickMath.sol
int24 internal constant MIN_TICK = -887272;
int24 internal constant MAX_TICK =  887272;

uint160 internal constant MIN_SQRT_RATIO = 4295128739;
uint160 internal constant MAX_SQRT_RATIO = 1461446703485210103287273052203988822378723970342;
```

These bounds correspond to prices of approximately 5.6×10^{-45} and 1.8×10^{44} respectively — effectively the entire representable price space.

5.2 Tick Spacing

Not every tick is eligible for use. Each fee tier enforces a **tick spacing** (`tickSpacing`), and positions must align their `tickLower` and `tickUpper` to multiples of `tickSpacing`. This prevents excessive tick fragmentation:

Fee Tier	Tick Spacing	Price Step per Initialized Tick
0.01% (100)	1	0.01%
0.05% (500)	10	0.10%
0.30% (3000)	60	0.60%
1.00% (10000)	200	2.00%

The tick spacing check is enforced in `mint()`:

```
require(tickLower < tickUpper, 'TLU');
require(tickLower >= TickMath.MIN_TICK, 'TLM');
require(tickUpper <= TickMath.MAX_TICK, 'TUM');
// alignment check performed via modulo in Tick.checkTicks()
```

5.3 `TickMath`: Tick ↔ sqrtPrice Conversion

`TickMath.sol` implements two pure functions using bitwise operations and lookup tables. The forward direction (`getSqrtRatioAtTick`) computes `1.0001^(i/2) · 2^96` using a sequence of up to 19 conditional multiplications by precomputed magic constants:

```
// TickMath.sol (simplified)
function getSqrtRatioAtTick(int24 tick) internal pure returns (uint160 sqrtPriceX96) {
    uint256 absTick = tick < 0 ? uint256(-int256(tick)) : uint256(int256(tick));
    require(absTick <= uint256(MAX_TICK), 'T');

    uint256 ratio = absTick & 0x1 != 0
        ? 0xfffcb933bd6fad37aa2d162d1a594001   // 2^128 / 1.0001^(1/2)
        : 0x100000000000000000000000000000000;  // 2^128

    if (absTick & 0x2  != 0) ratio = (ratio * 0xfff97272373d413259a46990580e213a) >> 128;
    if (absTick & 0x4  != 0) ratio = (ratio * 0xfff2e50f5f656932ef12357cf3c7fdcc) >> 128;
    if (absTick & 0x8  != 0) ratio = (ratio * 0xffe5caca7e10e4e61c3624eaa0941cd0) >> 128;
    if (absTick & 0x10 != 0) ratio = (ratio * 0xffcb9843d60f6159c9db58835c926644) >> 128;
    // ... continues for bits 0x20 through 0x80000
    if (absTick & 0x80000 != 0) ratio = (ratio * 0x48a170391f7dc42444e8fa2) >> 128;

    if (tick > 0) ratio = type(uint256).max / ratio;

    // convert from Q128 to Q64.96, rounding up
    sqrtPriceX96 = uint160((ratio >> 32) + (ratio % (1 << 32) == 0 ? 0 : 1));
}
```

The inverse direction (`getTickAtSqrtRatio`) uses a binary search approach with a log2 approximation:

```
function getTickAtSqrtRatio(uint160 sqrtPriceX96) internal pure returns (int24 tick) {
    require(sqrtPriceX96 >= MIN_SQRT_RATIO && sqrtPriceX96 < MAX_SQRT_RATIO, 'R');
    uint256 ratio = uint256(sqrtPriceX96) << 32;

    uint256 r = ratio;
    uint256 msb = 0;

    // compute floor(log2(ratio)) via bit manipulation
    assembly {
        let f := shl(7, gt(r, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))
        msb := or(msb, f)
        r := shr(f, r)
    }
    // ... 6 more assembly blocks for bits 64, 32, 16, 8, 4, 2
    assembly {
        r := shr(127, mul(r, r))
        let f := shr(128, r)
        log_2 := or(shl(51, f), log_2)
        r := shr(f, r)
    }

    int256 log_sqrt10001 = log_2 * 255738958999603826347141; // log base sqrt(1.0001) of 2

    int24 tickLow  = int24((log_sqrt10001 - 3402992956809132418596140100660247210) >> 128);
    int24 tickHigh = int24((log_sqrt10001 + 291339464771989622907027621153398088495) >> 128);

    tick = tickLow == tickHigh
        ? tickLow
        : getSqrtRatioAtTick(tickHigh) <= sqrtPriceX96
            ? tickHigh
            : tickLow;
}
```

---

6. Tick Bitmap: Efficient Range Navigation

6.1 Data Structure

The tick bitmap is a mapping from `int16 → uint256`. Each word (`uint256`) covers 256 consecutive ticks. For a pool with `tickSpacing = s`, each bit in word `w` represents tick `(w * 256 + bitPosition) * s`.

```
mapping(int16 => uint256) public override tickBitmap;
```

Key derivation:

```
// TickBitmap.sol
function position(int24 tick) private pure returns (int16 wordPos, uint8 bitPos) {
    wordPos = int16(tick >> 8);   // tick / 256
    bitPos  = uint8(tick % 256);  // tick % 256 (always positive, range [0,255])
}
```

Note: `tick` here refers to the **compressed tick** = `tick / tickSpacing`.

6.2 Flip Operation

When a tick transitions between initialized and uninitialized:

```
function flipTick(
    mapping(int16 => uint256) storage self,
    int24 tick,
    int24 tickSpacing
) internal {
    require(tick % tickSpacing == 0); // ensure on boundary
    (int16 wordPos, uint8 bitPos) = position(tick / tickSpacing);
    uint256 mask = 1 << bitPos;
    self[wordPos] ^= mask;  // XOR flips the bit
}
```

This is called during `mint()` when a tick first becomes initialized, and during `burn()` when `liquidityGross` drops to zero.

6.3 Next Initialized Tick Search

During a swap, the pool needs to find the next initialized tick to determine how far the price can move before crossing a boundary:

```
function nextInitializedTickWithinOneWord(
    mapping(int16 => uint256) storage self,
    int24  tick,
    int24  tickSpacing,
    bool   lte          // searching left (lte=true) or right (lte=false)
) internal view returns (int24 next, bool initialized) {
    int24 compressed = tick / tickSpacing;
    if (tick < 0 && tick % tickSpacing != 0) compressed--; // floor division for negatives

    if (lte) {
        (int16 wordPos, uint8 bitPos) = position(compressed);
        // create a mask of all bits <= bitPos
        uint256 mask = (1 << bitPos) - 1 + (1 << bitPos);
        uint256 masked = self[wordPos] & mask;

        initialized = masked != 0;
        // the next tick is the most significant bit of `masked`
        next = initialized
            ? (compressed - int24(bitPos - BitMath.mostSignificantBit(masked))) * tickSpacing
            : (compressed - int24(bitPos)) * tickSpacing;
    } else {
        // searching right (lte = false)
        (int16 wordPos, uint8 bitPos) = position(compressed + 1);
        uint256 mask = ~((1 << bitPos) - 1);
        uint256 masked = self[wordPos] & mask;

        initialized = masked != 0;
        next = initialized
            ? (compressed + 1 + int24(BitMath.leastSignificantBit(masked) - bitPos)) * tickSpacing
            : (compressed + 1 + int24(type(uint8).max - bitPos)) * tickSpacing;
    }
}
```

Critical limitation: The search only spans **one bitmap word (256 ticks)**. If no initialized tick exists in the current word, the search stops and returns the word boundary. The swap loop calls this function repeatedly until it finds an initialized tick or exhausts the price limit.

6.4 BitMath Library

```
// libraries/BitMath.sol
function mostSignificantBit(uint256 x) internal pure returns (uint8 r) {
    require(x > 0);
    if (x >= 0x100000000000000000000000000000000) { x >>= 128; r += 128; }
    if (x >= 0x10000000000000000)                 { x >>= 64;  r += 64;  }
    if (x >= 0x100000000)                         { x >>= 32;  r += 32;  }
    if (x >= 0x10000)                             { x >>= 16;  r += 16;  }
    if (x >= 0x100)                               { x >>= 8;   r += 8;   }
    if (x >= 0x10)                                { x >>= 4;   r += 4;   }
    if (x >= 0x4)                                 { x >>= 2;   r += 2;   }
    if (x >= 0x2)                                 r += 1;
}

function leastSignificantBit(uint256 x) internal pure returns (uint8 r) {
    require(x > 0);
    r = 255;
    if (x & type(uint128).max > 0) { r -= 128; } else { x >>= 128; }
    if (x & type(uint64).max  > 0) { r -= 64;  } else { x >>= 64;  }
    if (x & type(uint32).max  > 0) { r -= 32;  } else { x >>= 32;  }
    if (x & type(uint16).max  > 0) { r -= 16;  } else { x >>= 16;  }
    if (x & type(uint8).max   > 0) { r -= 8;   } else { x >>= 8;   }
    if (x & 0xf               > 0) { r -= 4;   } else { x >>= 4;   }
    if (x & 0x3               > 0) { r -= 2;   } else { x >>= 2;   }
    if (x & 0x1               > 0)  r -= 1;
}
```

7. Fixed-Point Math & sqrtPriceX96

7.1 Q64.96 Representation

`sqrtPriceX96` is a **Q64.96 fixed-point number**: the integer `sqrtPriceX96` represents the value `sqrtPriceX96 / 2^96`.

- Stored as `uint160` (20 bytes)

- Encodes the square root of the price: `sqrtPriceX96 = sqrt(token1/token0) · 2^96`
- Actual price: `price = (sqrtPriceX96)^2 / 2^192`

Why `sqrt(price)` and not `price` directly?

1. Square roots enable exact integer arithmetic for amount calculations (via `Δy = L · Δ√p` and `Δx = L · Δ(1/√p)`)
2. Q64.96 gives 64 bits for the integer part and 96 bits of precision — enough for the full `[MIN_TICK, MAX_TICK]` range

7.2 `FullMath` — 512-bit Multiplication Without Overflow

Standard 256-bit multiplication can overflow. `FullMath.sol` implements Knuth's algorithm for 512-bit intermediate results:

```
// libraries/FullMath.sol
function mulDiv(
    uint256 a,
    uint256 b,
    uint256 denominator
) internal pure returns (uint256 result) {
    // 512-bit multiply [prod1 prod0] = a * b
    uint256 prod0; // Least significant 256 bits of the product
    uint256 prod1; // Most significant 256 bits of the product
    assembly {
        let mm := mulmod(a, b, not(0))
        prod0 := mul(a, b)
        prod1 := sub(sub(mm, prod0), lt(mm, prod0))
    }

    // Handle cases where result fits in 256 bits (prod1 == 0)
    if (prod1 == 0) {
        require(denominator > 0);
        assembly { result := div(prod0, denominator) }
        return result;
    }

    require(denominator > prod1);

    // Make division exact by subtracting the remainder
    uint256 remainder;
    assembly { remainder := mulmod(a, b, denominator) }
    assembly { prod1 := sub(prod1, gt(remainder, prod0)) }
    assembly { prod0 := sub(prod0, remainder) }

    // Factor powers of two out of denominator
    uint256 twos = denominator & (~denominator + 1);
    assembly { denominator := div(denominator, twos) }
    assembly { prod0 := div(prod0, twos) }
    assembly { twos := add(div(sub(0, twos), twos), 1) }
    prod0 |= prod1 * twos;

    // Invert denominator mod 2^256 using Newton-Raphson
    uint256 inv = (3 * denominator) ^ 2;
    inv *= 2 - denominator * inv;
    inv *= 2 - denominator * inv;
    inv *= 2 - denominator * inv;
    inv *= 2 - denominator * inv;
    inv *= 2 - denominator * inv;
    inv *= 2 - denominator * inv;

    result = prod0 * inv;
}
```

This is invoked throughout `SqrtPriceMath`, `SwapMath`, and `LiquidityMath` wherever intermediate products could exceed 256 bits.

7.3 Fixed-Point Notation Summary

Type	Scale	Library	Used For
Q64.96	`2^96`	`FixedPoint96.sol`	`sqrtPriceX96`

Type	Scale	Library	Used For
Q128.128	`2^128`	`FixedPoint128.sol`	`feeGrowthGlobal{0,1}X128`, `feeGrowthInside`
Q128	`2^128`	`FullMath.sol`	`secondsPerLiquidityCumulativeX128`
UQ112x112	`2^112`	(v2 only)	Not used in v3

8. Factory: Pool Deployment

8.1 Fee Amount Registration

The factory maintains a registry of fee tiers and their associated tick spacings:

```
// UniswapV3Factory.sol
mapping(uint24 => int24) public override feeAmountTickSpacing;

constructor() {
    owner = msg.sender;
    emit OwnerChanged(address(0), msg.sender);

    feeAmountTickSpacing[500]   = 10;
    feeAmountTickSpacing[3000]  = 60;
    feeAmountTickSpacing[10000] = 200;
}

function enableFeeAmount(uint24 fee, int24 tickSpacing) public override {
    require(msg.sender == owner);
    require(fee < 1000000);         // fee < 100%
    require(tickSpacing > 0 && tickSpacing < 16384);
    require(feeAmountTickSpacing[fee] == 0);  // not already set

    feeAmountTickSpacing[fee] = tickSpacing;
    emit FeeAmountEnabled(fee, tickSpacing);
}
```

After deployment, the fee tier 100 (0.01%, tickSpacing=1) was added via governance for stable pairs.

8.2 CREATE2 Pool Deployment

```
// UniswapV3Factory.sol
function createPool(
    address tokenA,
    address tokenB,
    uint24  fee
) external override noDelegateCall returns (address pool) {
    require(tokenA != tokenB);
    (address token0, address token1) = tokenA < tokenB
        ? (tokenA, tokenB)
        : (tokenB, tokenA);
    require(token0 != address(0));
    int24 tickSpacing = feeAmountTickSpacing[fee];
    require(tickSpacing != 0);
    require(getPool[token0][token1][fee] == address(0));

    pool = deploy(address(this), token0, token1, fee, tickSpacing);
    getPool[token0][token1][fee]  = pool;
    getPool[token1][token0][fee]  = pool;  // also index reverse direction
    emit PoolCreated(token0, token1, fee, tickSpacing, pool);
}
```

The `deploy()` helper uses assembly for `CREATE2`:

```
// UniswapV3PoolDeployer.sol
function deploy(
    address factory,
    address token0,
    address token1,
    uint24  fee,
    int24   tickSpacing
) internal returns (address pool) {
    parameters = Parameters({factory: factory, token0: token0, token1: token1, fee: fee, tickSpacing:
tickSpacing});
    pool = address(new UniswapV3Pool{salt: keccak256(abi.encode(token0, token1, fee))}());
    delete parameters;
}
```

The `parameters` transient storage trick passes constructor arguments to the pool without constructor calldata — the pool reads them from the deployer via `IUniswapV3PoolDeployer(msg.sender).parameters()`.

8.3 Pool Address Computation (Off-chain)

```
// PoolAddress.sol (periphery)
function computeAddress(address factory, PoolKey memory key)
    internal pure returns (address pool)
{
    require(key.token0 < key.token1);
    pool = address(
        uint256(
            keccak256(abi.encodePacked(
                hex'ff',
                factory,
                keccak256(abi.encode(key.token0, key.token1, key.fee)),
                POOL_INIT_CODE_HASH  // = 0xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54
            ))
        )
    );
}
```

The `POOL_INIT_CODE_HASH` is the keccak256 of the pool's creation bytecode, enabling counterfactual address computation.

---

9. Pool: Swap Mechanics

9.1 Swap Function Signature

```
// UniswapV3Pool.sol
function swap(
    address recipient,
    bool    zeroForOne,        // true = token0→token1, false = token1→token0
    int256  amountSpecified,   // positive = exact input, negative = exact output
    uint160 sqrtPriceLimitX96, // price boundary (slippage protection)
    bytes   calldata data      // callback data
) external override noDelegateCall returns (int256 amount0, int256 amount1)
```

9.2 Swap State Machine

The swap is implemented as a loop over tick ranges. At each step, the pool computes how far the price can move within the current tick range before hitting a tick boundary:

```
// Internal state for the swap
struct SwapState {
    int256   amountSpecifiedRemaining;  // amount left to fill
    int256   amountCalculated;          // cumulative output (or input in exact-output)
    uint160  sqrtPriceX96;             // current sqrt price
    int24    tick;                      // current tick
    uint256  feeGrowthGlobalX128;      // global fee accumulator (token in direction)
    uint128  protocolFee;              // accumulated protocol fee
    uint128  liquidity;                // current active liquidity
}

struct StepComputations {
    uint160  sqrtPriceStartX96;        // sqrt price at beginning of step
    int24    tickNext;                  // next initialized tick
    bool     initialized;              // whether tickNext is initialized
    uint160  sqrtPriceNextX96;         // sqrt price at tickNext
    uint256  amountIn;                 // input for this step
    uint256  amountOut;                // output for this step
    uint256  feeAmount;                // fee charged for this step
}
```

9.3 Full Swap Loop

```
// UniswapV3Pool.sol — swap() internal logic (condensed)

// Acquire reentrancy lock
require(slot0Start.unlocked, 'LOK');
slot0.unlocked = false;

SwapState memory state = SwapState({
    amountSpecifiedRemaining: amountSpecified,
    amountCalculated:         0,
    sqrtPriceX96:            slot0Start.sqrtPriceX96,
    tick:                    slot0Start.tick,
    feeGrowthGlobalX128:     zeroForOne ? feeGrowthGlobal0X128 : feeGrowthGlobal1X128,
    protocolFee:             0,
    liquidity:               liquidityStart
});

// Main swap loop
while (state.amountSpecifiedRemaining != 0 && state.sqrtPriceX96 != sqrtPriceLimitX96) {
    StepComputations memory step;
    step.sqrtPriceStartX96 = state.sqrtPriceX96;

    // 1. Find next initialized tick in current bitmap word
    (step.tickNext, step.initialized) = tickBitmap.nextInitializedTickWithinOneWord(
        state.tick,
        tickSpacing,
        zeroForOne
    );

    // 2. Clamp to valid tick range
    if (step.tickNext < TickMath.MIN_TICK) step.tickNext = TickMath.MIN_TICK;
    else if (step.tickNext > TickMath.MAX_TICK) step.tickNext = TickMath.MAX_TICK;

    step.sqrtPriceNextX96 = TickMath.getSqrtRatioAtTick(step.tickNext);

    // 3. Compute swap within current tick range
    (state.sqrtPriceX96, step.amountIn, step.amountOut, step.feeAmount) = SwapMath.computeSwapStep(
        state.sqrtPriceX96,
        // apply price limit: can't cross sqrtPriceLimitX96
        (zeroForOne
            ? step.sqrtPriceNextX96 < sqrtPriceLimitX96
            : step.sqrtPriceNextX96 > sqrtPriceLimitX96
        ) ? sqrtPriceLimitX96 : step.sqrtPriceNextX96,
        state.liquidity,
        state.amountSpecifiedRemaining,
        fee
    );

    // 4. Update accounting
    if (exactInput) {
        state.amountSpecifiedRemaining -= (step.amountIn + step.feeAmount).toInt256();
        state.amountCalculated         = state.amountCalculated.sub(step.amountOut.toInt256());
    } else {
        state.amountSpecifiedRemaining += step.amountOut.toInt256();
        state.amountCalculated          = state.amountCalculated.add(
            (step.amountIn + step.feeAmount).toInt256()
        );
    }

    // 5. Handle protocol fee
    if (cache.feeProtocol > 0) {
        uint256 delta = step.feeAmount / cache.feeProtocol;
        step.feeAmount      -= delta;
        state.protocolFee   += uint128(delta);
    }

    // 6. Update global fee accumulator
    if (state.liquidity > 0) {
        state.feeGrowthGlobalX128 += FullMath.mulDiv(
            step.feeAmount,
            FixedPoint128.Q128,
            state.liquidity
        );
    }

    // 7. Cross tick if price reached the boundary
    if (state.sqrtPriceX96 == step.sqrtPriceNextX96) {
        if (step.initialized) {
            // Write oracle observation before crossing
            (int56 tickCumulative, uint160 secondsPerLiquidityCumulativeX128) = observations.observeSingle(
                cache.blockTimestamp, 0, slot0Start.tick, slot0Start.observationIndex,
                liquidityStart, slot0Start.observationCardinality
            );

            int128 liquidityNet = ticks.cross(
                step.tickNext,
                (zeroForOne ? state.feeGrowthGlobalX128 : feeGrowthGlobal0X128),
                (zeroForOne ? feeGrowthGlobal1X128 : state.feeGrowthGlobalX128),
                secondsPerLiquidityCumulativeX128,
                tickCumulative,
                cache.blockTimestamp
            );

            // Flip sign if going right→left
            if (zeroForOne) liquidityNet = -liquidityNet;
            state.liquidity = LiquidityMath.addDelta(state.liquidity, liquidityNet);
        }

        state.tick = zeroForOne ? step.tickNext - 1 : step.tickNext;
    } else if (state.sqrtPriceX96 != step.sqrtPriceStartX96) {
        // Recompute tick from new sqrtPrice (price moved within a word, no tick crossed)
        state.tick = TickMath.getTickAtSqrtRatio(state.sqrtPriceX96);
    }
}

// Write final state back to storage
if (state.tick != slot0Start.tick) {
    // write oracle observation
    (uint16 observationIndex, uint16 observationCardinality) = observations.write(
        slot0Start.observationIndex, cache.blockTimestamp, slot0Start.tick,
        liquidityStart, slot0Start.observationCardinality, slot0Start.observationCardinalityNext
    );
    (slot0.sqrtPriceX96, slot0.tick, slot0.observationIndex, slot0.observationCardinality) =
        (state.sqrtPriceX96, state.tick, observationIndex, observationCardinality);
} else {
    slot0.sqrtPriceX96 = state.sqrtPriceX96;
}

if (liquidityStart != state.liquidity) liquidity = state.liquidity;

// Update fee growth globals
if (zeroForOne) {
    feeGrowthGlobal0X128 = state.feeGrowthGlobalX128;
    if (state.protocolFee > 0) protocolFees.token0 += state.protocolFee;
} else {
    feeGrowthGlobal1X128 = state.feeGrowthGlobalX128;
    if (state.protocolFee > 0) protocolFees.token1 += state.protocolFee;
```

```
}

// Compute net token transfers (amounts can be positive or negative)
(amount0, amount1) = zeroForOne == exactInput
    ? (amountSpecified - state.amountSpecifiedRemaining, state.amountCalculated)
    : (state.amountCalculated, amountSpecified - state.amountSpecifiedRemaining);

// Transfer tokens + invoke callback
if (zeroForOne) {
    if (amount1 < 0) TransferHelper.safeTransfer(token1, recipient, uint256(-amount1));
    uint256 balance0Before = balance0();
    IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
    require(balance0Before.add(uint256(amount0)) <= balance0(), 'IIA');
} else {
    if (amount0 < 0) TransferHelper.safeTransfer(token0, recipient, uint256(-amount0));
    uint256 balance1Before = balance1();
    IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
    require(balance1Before.add(uint256(amount1)) <= balance1(), 'IIA');
}

slot0.unlocked = true;
```

9.4 `SwapMath.computeSwapStep`

This function computes the result of a single swap step within a tick range:

```
// libraries/SwapMath.sol
function computeSwapStep(
    uint160 sqrtRatioCurrentX96,
    uint160 sqrtRatioTargetX96,   // either tickNext or sqrtPriceLimitX96
    uint128 liquidity,
    int256  amountRemaining,
    uint24  feePips
) internal pure returns (
    uint160 sqrtRatioNextX96,
    uint256 amountIn,
    uint256 amountOut,
    uint256 feeAmount
) {
    bool zeroForOne = sqrtRatioCurrentX96 >= sqrtRatioTargetX96;
    bool exactIn    = amountRemaining >= 0;

    if (exactIn) {
        uint256 amountRemainingLessFee = FullMath.mulDiv(
            uint256(amountRemaining), 1e6 - feePips, 1e6
        );
        // Compute how far we can move the price with amountRemainingLessFee as input
        amountIn = zeroForOne
            ? SqrtPriceMath.getAmount0Delta(sqrtRatioTargetX96, sqrtRatioCurrentX96, liquidity, true)
            : SqrtPriceMath.getAmount1Delta(sqrtRatioCurrentX96, sqrtRatioTargetX96, liquidity, true);

        if (amountRemainingLessFee >= amountIn) {
            // Enough to reach target
            sqrtRatioNextX96 = sqrtRatioTargetX96;
        } else {
            // Can only partially fill — compute new price from actual input
            sqrtRatioNextX96 = SqrtPriceMath.getNextSqrtPriceFromInput(
                sqrtRatioCurrentX96, liquidity, amountRemainingLessFee, zeroForOne
            );
        }
    } else {
        // exact output
        amountOut = zeroForOne
            ? SqrtPriceMath.getAmount1Delta(sqrtRatioTargetX96, sqrtRatioCurrentX96, liquidity, false)
            : SqrtPriceMath.getAmount0Delta(sqrtRatioCurrentX96, sqrtRatioTargetX96, liquidity, false);

        if (uint256(-amountRemaining) >= amountOut) {
            sqrtRatioNextX96 = sqrtRatioTargetX96;
        } else {
            sqrtRatioNextX96 = SqrtPriceMath.getNextSqrtPriceFromOutput(
                sqrtRatioCurrentX96, liquidity, uint256(-amountRemaining), zeroForOne
            );
        }
    }

    bool max = sqrtRatioTargetX96 == sqrtRatioNextX96;

    if (zeroForOne) {
        amountIn  = max && exactIn ? amountIn : SqrtPriceMath.getAmount0Delta(sqrtRatioNextX96,
sqrtRatioCurrentX96, liquidity, true);
        amountOut = max && !exactIn ? amountOut : SqrtPriceMath.getAmount1Delta(sqrtRatioNextX96,
sqrtRatioCurrentX96, liquidity, false);
    } else {
        amountIn  = max && exactIn ? amountIn : SqrtPriceMath.getAmount1Delta(sqrtRatioCurrentX96,
sqrtRatioNextX96, liquidity, true);
        amountOut = max && !exactIn ? amountOut : SqrtPriceMath.getAmount0Delta(sqrtRatioCurrentX96,
sqrtRatioNextX96, liquidity, false);
    }

    // Cap output if exact output (can't output more than requested)
    if (!exactIn && amountOut > uint256(-amountRemaining)) {
        amountOut = uint256(-amountRemaining);
    }

    // Fee is taken on top of amountIn in exact-input mode
    if (exactIn && sqrtRatioNextX96 != sqrtRatioTargetX96) {
        feeAmount = uint256(amountRemaining) - amountIn;
    } else {
        feeAmount = FullMath.mulDivRoundingUp(amountIn, feePips, 1e6 - feePips);
    }
}
```

---

10. Pool: Liquidity Provisioning (Mint & Burn)

10.1 `mint()` — Add Liquidity

```
// UniswapV3Pool.sol
function mint(
    address recipient,
    int24   tickLower,
    int24   tickUpper,
    uint128 amount,       // liquidity units to add
    bytes   calldata data // callback data
) external override lock noDelegateCall returns (uint256 amount0, uint256 amount1) {
    require(amount > 0);

    (, int256 amount0Int, int256 amount1Int) = _updatePosition(
        recipient, tickLower, tickUpper, int256(amount).toInt128(), slot0.tick
    );

    amount0 = uint256(amount0Int);
    amount1 = uint256(amount1Int);

    uint256 balance0Before;
    uint256 balance1Before;
    if (amount0 > 0) balance0Before = balance0();
    if (amount1 > 0) balance1Before = balance1();

    // Optimistic transfer — call the callback, expecting tokens to arrive
    IUniswapV3MintCallback(msg.sender).uniswapV3MintCallback(amount0, amount1, data);

    if (amount0 > 0) require(balance0Before.add(amount0) <= balance0(), 'M0');
    if (amount1 > 0) require(balance1Before.add(amount1) <= balance1(), 'M1');

    emit Mint(msg.sender, recipient, tickLower, tickUpper, amount, amount0, amount1);
}
```

10.2 `_updatePosition()` — Core Position Update

```
function _updatePosition(
    address owner,
    int24   tickLower,
    int24   tickUpper,
    int128  liquidityDelta,   // positive = add, negative = remove
    int24   tick              // current tick
) private returns (Position.Info storage position, int256 amount0, int256 amount1) {
    position = positions.get(owner, tickLower, tickUpper);

    uint256 _feeGrowthGlobal0X128 = feeGrowthGlobal0X128;
    uint256 _feeGrowthGlobal1X128 = feeGrowthGlobal1X128;

    // if we need to update the ticks, do it
    bool flippedLower;
    bool flippedUpper;
    if (liquidityDelta != 0) {
        uint32 blockTimestamp = uint32(block.timestamp);
        (int56 tickCumulative, uint160 secondsPerLiquidityCumulativeX128) =
            observations.observeSingle(blockTimestamp, 0, slot0.tick, slot0.observationIndex,
                liquidity, slot0.observationCardinality);

        flippedLower = ticks.update(
            tickLower, tick, liquidityDelta,
            _feeGrowthGlobal0X128, _feeGrowthGlobal1X128,
            secondsPerLiquidityCumulativeX128, tickCumulative,
            blockTimestamp, false, maxLiquidityPerTick
        );
        flippedUpper = ticks.update(
            tickUpper, tick, liquidityDelta,
            _feeGrowthGlobal0X128, _feeGrowthGlobal1X128,
            secondsPerLiquidityCumulativeX128, tickCumulative,
            blockTimestamp, true, maxLiquidityPerTick
        );

        if (flippedLower) tickBitmap.flipTick(tickLower, tickSpacing);
        if (flippedUpper) tickBitmap.flipTick(tickUpper, tickSpacing);
    }

    // Compute fee growth inside the range
    (uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) =
        ticks.getFeeGrowthInside(tickLower, tickUpper, tick,
            _feeGrowthGlobal0X128, _feeGrowthGlobal1X128);

    // Update position fee accumulators and liquidity
    position.update(liquidityDelta, feeGrowthInside0X128, feeGrowthInside1X128);

    // Remove ticks from bitmap if they're no longer needed
    if (liquidityDelta < 0) {
        if (flippedLower) ticks.clear(tickLower);
        if (flippedUpper) ticks.clear(tickUpper);
    }

    // Compute actual token amounts required/returned
    if (liquidityDelta != 0) {
        if (tick < tickLower) {
            // Position entirely above current price — only token0
            amount0 = SqrtPriceMath.getAmount0Delta(
                TickMath.getSqrtRatioAtTick(tickLower),
                TickMath.getSqrtRatioAtTick(tickUpper),
                liquidityDelta
            );
        } else if (tick < tickUpper) {
            // Position spans current price — both tokens
            uint128 liquidityBefore = liquidity;
            (uint16 observationIndex, uint16 observationCardinality) = observations.write(...);
            slot0.observationIndex    = observationIndex;
            slot0.observationCardinality = observationCardinality;
            amount0 = SqrtPriceMath.getAmount0Delta(
                slot0.sqrtPriceX96,
                TickMath.getSqrtRatioAtTick(tickUpper),
                liquidityDelta
            );
            amount1 = SqrtPriceMath.getAmount1Delta(
                TickMath.getSqrtRatioAtTick(tickLower),
                slot0.sqrtPriceX96,
                liquidityDelta
            );

            // Update active liquidity
            liquidity = LiquidityMath.addDelta(liquidityBefore, liquidityDelta);
        } else {
            // Position entirely below current price — only token1
            amount1 = SqrtPriceMath.getAmount1Delta(
                TickMath.getSqrtRatioAtTick(tickLower),
                TickMath.getSqrtRatioAtTick(tickUpper),
                liquidityDelta
            );
        }
    }
}
```

10.3 `burn()` — Remove Liquidity

```
function burn(
    int24   tickLower,
    int24   tickUpper,
    uint128 amount
) external override lock noDelegateCall returns (uint256 amount0, uint256 amount1) {
    // liquidityDelta is negative for burn
    (Position.Info storage position, int256 amount0Int, int256 amount1Int) =
        _updatePosition(msg.sender, tickLower, tickUpper, -int256(amount).toInt128(), slot0.tick);

    amount0 = uint256(-amount0Int);
    amount1 = uint256(-amount1Int);

    if (amount0 > 0 || amount1 > 0) {
        // Accrue to position's owed balances (NOT transferred here)
        (position.tokensOwed0, position.tokensOwed1) = (
            position.tokensOwed0 + uint128(amount0),
            position.tokensOwed1 + uint128(amount1)
        );
    }

    emit Burn(msg.sender, tickLower, tickUpper, amount, amount0, amount1);
}
```

10.4 `collect()` — Withdraw Tokens and Fees

Tokens burned by `burn()` and accumulated fees are NOT transferred automatically. The LP must call `collect()` separately:

```
function collect(
    address recipient,
    int24   tickLower,
    int24   tickUpper,
    uint128 amount0Requested,
    uint128 amount1Requested
) external override lock returns (uint128 amount0, uint128 amount1) {
    Position.Info storage position = positions.get(msg.sender, tickLower, tickUpper);

    amount0 = amount0Requested > position.tokensOwed0 ? position.tokensOwed0 : amount0Requested;
    amount1 = amount1Requested > position.tokensOwed1 ? position.tokensOwed1 : amount1Requested;
    if (amount0 > 0) {
        position.tokensOwed0 -= amount0;
        TransferHelper.safeTransfer(token0, recipient, amount0);
    }
    if (amount1 > 0) {
        position.tokensOwed1 -= amount1;
        TransferHelper.safeTransfer(token1, recipient, amount1);
    }

    emit Collect(msg.sender, recipient, tickLower, tickUpper, amount0, amount1);
}
```

The two-step `burn()` + `collect()` pattern separates accounting from transfer, which is important for the NFT position wrapper.

---

11. Fee Accounting System

11.1 Fee Tiers

Each pool is deployed with a fixed fee tier, expressed in units of **hundredths of a basis point** (1e-6):

Fee Value	Fee Percentage	Tick Spacing	Typical Use
100	0.01%	1	Stablecoin pairs (USDC/USDT)
500	0.05%	10	Pegged assets (WBTC/ETH)
3000	0.30%	60	Standard volatile pairs
10000	1.00%	200	Exotic/long-tail assets

11.2 Fee Growth Per Liquidity (Global Accumulators)

The pool tracks fees earned per unit of liquidity as **global, monotonically increasing** accumulators in Q128.128 format:

```
feeGrowthGlobal0X128 += feeAmount0 * Q128 / liquidity
feeGrowthGlobal1X128 += feeAmount1 * Q128 / liquidity
```

This is updated in `SwapMath.computeSwapStep()` for every swap step.

11.3 Fee Growth Inside a Range

For a specific tick range `[tickLower, tickUpper]`, the fee growth **inside** the range is computed using the outside accumulators stored in each tick:

```
// Tick.sol
function getFeeGrowthInside(
    mapping(int24 => Tick.Info) storage self,
    int24   tickLower,
    int24   tickUpper,
    int24   tickCurrent,
    uint256 feeGrowthGlobal0X128,
    uint256 feeGrowthGlobal1X128
) internal view returns (uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) {
    Info storage lower = self[tickLower];
    Info storage upper = self[tickUpper];

    // Determine the fee growth below tickLower
    uint256 feeGrowthBelow0X128;
    uint256 feeGrowthBelow1X128;
    if (tickCurrent >= tickLower) {
        feeGrowthBelow0X128 = lower.feeGrowthOutside0X128;
        feeGrowthBelow1X128 = lower.feeGrowthOutside1X128;
    } else {
        feeGrowthBelow0X128 = feeGrowthGlobal0X128 - lower.feeGrowthOutside0X128;
        feeGrowthBelow1X128 = feeGrowthGlobal1X128 - lower.feeGrowthOutside1X128;
    }
    // Determine the fee growth above tickUpper
    uint256 feeGrowthAbove0X128;
    uint256 feeGrowthAbove1X128;
    if (tickCurrent < tickUpper) {
        feeGrowthAbove0X128 = upper.feeGrowthOutside0X128;
        feeGrowthAbove1X128 = upper.feeGrowthOutside1X128;
    } else {
        feeGrowthAbove0X128 = feeGrowthGlobal0X128 - upper.feeGrowthOutside0X128;
        feeGrowthAbove1X128 = feeGrowthGlobal1X128 - upper.feeGrowthOutside1X128;
    }

    feeGrowthInside0X128 = feeGrowthGlobal0X128 - feeGrowthBelow0X128 - feeGrowthAbove0X128;
    feeGrowthInside1X128 = feeGrowthGlobal1X128 - feeGrowthBelow1X128 - feeGrowthAbove1X128;
}
```

11.4 Position Fee Accrual

When a position is updated (by calling `mint`, `burn`, or `collect` via `_updatePosition`), fees are settled:

```
// Position.sol
function update(
    Info storage self,
    int128  liquidityDelta,
    uint256 feeGrowthInside0X128,
    uint256 feeGrowthInside1X128
) internal {
    Info memory _self = self;

    uint128 liquidityNext;
    if (liquidityDelta == 0) {
        require(_self.liquidity > 0, 'NP'); // disallow pokes for 0 liquidity positions
        liquidityNext = _self.liquidity;
    } else {
        liquidityNext = LiquidityMath.addDelta(_self.liquidity, liquidityDelta);
    }

    // Calculate fees earned (using overflow-safe subtraction for Q128.128 accumulators)
    uint128 tokensOwed0 = uint128(
        FullMath.mulDiv(
            feeGrowthInside0X128 - _self.feeGrowthInside0LastX128,
            _self.liquidity,
            FixedPoint128.Q128
        )
    );
    uint128 tokensOwed1 = uint128(
        FullMath.mulDiv(
            feeGrowthInside1X128 - _self.feeGrowthInside1LastX128,
            _self.liquidity,
            FixedPoint128.Q128
        )
    );

    if (liquidityDelta != 0) self.liquidity = liquidityNext;
    self.feeGrowthInside0LastX128 = feeGrowthInside0X128;
    self.feeGrowthInside1LastX128 = feeGrowthInside1X128;
    if (tokensOwed0 > 0 || tokensOwed1 > 0) {
        self.tokensOwed0 += tokensOwed0;
        self.tokensOwed1 += tokensOwed1;
    }
}
```

The subtraction `feeGrowthInside0X128 - _self.feeGrowthInside0LastX128` is performed in modular arithmetic (uint256 overflow is intentional), correctly handling the case where the global accumulator wraps around.

11.5 Protocol Fee

A portion of the swap fee can be directed to a protocol-controlled address. The protocol fee is configured per pool via `setFeeProtocol(feeProtocol0, feeProtocol1)`:

```
function setFeeProtocol(uint8 feeProtocol0, uint8 feeProtocol1) external override lock onlyFactoryOwner {
    require(
        (feeProtocol0 == 0 || (feeProtocol0 >= 4 && feeProtocol0 <= 10)) &&
        (feeProtocol1 == 0 || (feeProtocol1 >= 4 && feeProtocol1 <= 10))
    );
    uint8 feeProtocolOld = slot0.feeProtocol;
    slot0.feeProtocol = feeProtocol0 + (feeProtocol1 << 4);
    emit SetFeeProtocol(feeProtocolOld % 16, feeProtocolOld >> 4, feeProtocol0, feeProtocol1);
}
```

The `feeProtocol` value represents the **denominator** of the fraction taken from the swap fee. If `feeProtocol0 = 5`, the protocol receives `1/5 = 20%` of the token0 swap fees. Valid values: 0 (disabled) or 4–10 (10%–25%).

During each swap step:

```
uint256 delta = step.feeAmount / cache.feeProtocol;
step.feeAmount -= delta;   // LP fee is reduced
state.protocolFee += uint128(delta);
```

Collected via `collectProtocol()` (owner only).

---

12. Price Oracle (Observations Array)

12.1 Architecture

V3's oracle stores up to 65,535 observations in a **pre-allocated ring buffer**. Unlike v2's always-on TWAP, v3 observations are written only when the tick changes (on the first swap of each block). The cardinality of the ring buffer is set to 1 by default and expanded on demand.

```
// Oracle.sol
struct Observation {
    uint32  blockTimestamp;
    int56   tickCumulative;                      // sum of (tick * seconds) — arithmetic TWAP
    uint160 secondsPerLiquidityCumulativeX128;  // sum of (1/L * seconds) — liquidity-weighted
    bool    initialized;
}
```

12.2 Writing Observations

```
function write(
    Observation[65535] storage self,
    uint16  index,       // current ring buffer write index
    uint32  blockTimestamp,
    int24   tick,
    uint128 liquidity,
    uint16  cardinality,
    uint16  cardinalityNext
) internal returns (uint16 indexUpdated, uint16 cardinalityUpdated) {
    Observation memory last = self[index];

    // Only write at most once per block
    if (last.blockTimestamp == blockTimestamp) return (index, cardinality);

    // Expand cardinality if needed
    if (cardinalityNext > cardinality && indexUpdated == (cardinality - 1)) {
        cardinalityUpdated = cardinalityNext;
    } else {
        cardinalityUpdated = cardinality;
    }
    indexUpdated = (index + 1) % cardinalityUpdated;
    self[indexUpdated] = transform(last, blockTimestamp, tick, liquidity);
}

function transform(
    Observation memory last,
    uint32  blockTimestamp,
    int24   tick,
    uint128 liquidity
) private pure returns (Observation memory) {
    uint32 delta = blockTimestamp - last.blockTimestamp;
    return Observation({
        blockTimestamp: blockTimestamp,
        tickCumulative: last.tickCumulative + int56(tick) * delta,
        secondsPerLiquidityCumulativeX128: last.secondsPerLiquidityCumulativeX128 +
            ((uint160(delta) << 128) / (liquidity > 0 ? liquidity : 1)),
        initialized: true
    });
}
```

12.3 Reading TWAP Values

```
function observe(
    Observation[65535] storage self,
    uint32   time,
    uint32[] memory secondsAgos,    // array of lookback periods
    int24    tick,
    uint16   index,
    uint128  liquidity,
    uint16   cardinality
) internal view returns (
    int56[]   memory tickCumulatives,
    uint160[] memory secondsPerLiquidityCumulativeX128s
) {
    for (uint256 i = 0; i < secondsAgos.length; i++) {
        (tickCumulatives[i], secondsPerLiquidityCumulativeX128s[i]) =
            observeSingle(self, time, secondsAgos[i], tick, index, liquidity, cardinality);
    }
}
```

Computing a TWAP price from two observations:

```
// Off-chain computation:
int56 tickCumulativeDelta = tickCumulative1 - tickCumulative0;
uint32 timeDelta          = blockTimestamp1 - blockTimestamp0;
int24  arithmeticMeanTick = int24(tickCumulativeDelta / timeDelta);

// Convert to price:
// price = 1.0001^arithmeticMeanTick
```

The geometric mean price is recovered from the arithmetic mean of `log(price)` ticks:

```
geometric_mean_price = 1.0001^(Σ(tick_i * Δt_i) / Σ(Δt_i))
```

This is manipulation-resistant because an attacker must hold an extreme price for an entire block to influence even a single observation — and the TWAP averages over many blocks.

12.4 Increasing Oracle Cardinality

```
function increaseObservationCardinalityNext(
    uint16 observationCardinalityNext
) external override lock noDelegateCall {
    uint16 observationCardinalityNextOld = slot0.observationCardinalityNext;
    uint16 observationCardinalityNextNew = observations.grow(
        observationCardinalityNextOld,
        observationCardinalityNext
    );
    slot0.observationCardinalityNext = observationCardinalityNextNew;
    if (observationCardinalityNextOld != observationCardinalityNextNew)
        emit IncreaseObservationCardinalityNext(observationCardinalityNextOld, observationCardinalityNextNew);
}
```

The `grow()` function pre-initializes new `Observation` slots (setting `blockTimestamp = 1`, `initialized = false`) so their first use requires only a warm SSTORE (5,000 gas) rather than a cold one (20,000 gas).

---

13. Flash Loans

V3 introduces a dedicated `flash()` function (separate from swaps, unlike v2 flash swaps):

```
// UniswapV3Pool.sol
function flash(
    address recipient,
    uint256 amount0,
    uint256 amount1,
    bytes   calldata data
) external override lock noDelegateCall {
    uint128 _liquidity = liquidity;
    require(_liquidity > 0, 'L');

    // Compute fees owed on the borrowed amounts
    uint256 fee0 = FullMath.mulDivRoundingUp(amount0, fee, 1e6);
    uint256 fee1 = FullMath.mulDivRoundingUp(amount1, fee, 1e6);

    uint256 balance0Before = balance0();
    uint256 balance1Before = balance1();

    // Transfer tokens optimistically
    if (amount0 > 0) TransferHelper.safeTransfer(token0, recipient, amount0);
    if (amount1 > 0) TransferHelper.safeTransfer(token1, recipient, amount1);

    // Invoke callback — recipient must repay amount + fee
    IUniswapV3FlashCallback(msg.sender).uniswapV3FlashCallback(fee0, fee1, data);

    uint256 balance0After = balance0();
    uint256 balance1After = balance1();

    require(balance0Before.add(fee0) <= balance0After, 'F0');
    require(balance1Before.add(fee1) <= balance1After, 'F1');

    // Distribute fees between protocol and LPs
    uint256 paid0 = balance0After - balance0Before;
    uint256 paid1 = balance1After - balance1Before;

    if (paid0 > 0) {
        uint8 feeProtocol0 = slot0.feeProtocol % 16;
        uint256 pFees0 = feeProtocol0 == 0 ? 0 : paid0 / feeProtocol0;
        if (pFees0 > 0) protocolFees.token0 += uint128(pFees0);
        feeGrowthGlobal0X128 += FullMath.mulDiv(paid0 - pFees0, FixedPoint128.Q128, _liquidity);
    }
    if (paid1 > 0) {
        uint8 feeProtocol1 = slot0.feeProtocol >> 4;
        uint256 pFees1 = feeProtocol1 == 0 ? 0 : paid1 / feeProtocol1;
        if (pFees1 > 0) protocolFees.token1 += uint128(pFees1);
        feeGrowthGlobal1X128 += FullMath.mulDiv(paid1 - pFees1, FixedPoint128.Q128, _liquidity);
    }

    emit Flash(msg.sender, recipient, amount0, amount1, paid0, paid1);
}
```

Key distinctions from v2 flash swaps:

- Flash fees accrue to LPs (via `feeGrowthGlobal`), not the pool's reserve
- Can borrow both tokens simultaneously
- Callback is `uniswapV3FlashCallback` (not embedded in swap callback)
- Fee calculation uses `mulDivRoundingUp` (borrower always pays at least 1 wei)

---

14. Periphery: NonfungiblePositionManager

14.1 Role and Architecture

`NonfungiblePositionManager` (NFPM) is the user-facing LP interface. It wraps each liquidity position as an **ERC-721 NFT**, enabling positions to be traded, transferred, and composited in DeFi. From the pool's perspective, NFPM is the single position owner for all positions it manages.

```
// NonfungiblePositionManager.sol
struct Position {
    uint96  nonce;          // prevents signature replay in permit
    address operator;       // approved address
    uint80  poolId;         // index into _poolIdToPoolKey mapping (saves space)
    int24   tickLower;
    int24   tickUpper;
    uint128 liquidity;
    uint256 feeGrowthInside0LastX128;
    uint256 feeGrowthInside1LastX128;
    uint128 tokensOwed0;
    uint128 tokensOwed1;
}

mapping(uint256 => Position)      private _positions;
mapping(uint80  => PoolAddress.PoolKey) private _poolIdToPoolKey;
mapping(address => uint80)        private _poolIds;
```

14.2 Minting an NFT Position

```
struct MintParams {
    address token0;
    address token1;
    uint24  fee;
    int24   tickLower;
    int24   tickUpper;
    uint256 amount0Desired;
    uint256 amount1Desired;
    uint256 amount0Min;    // slippage bound
    uint256 amount1Min;    // slippage bound
    address recipient;
    uint256 deadline;
}

function mint(MintParams calldata params)
    external payable override checkDeadline(params.deadline)
    returns (
        uint256 tokenId,
        uint128 liquidity,
        uint256 amount0,
        uint256 amount1
    )
{
    IUniswapV3Pool pool;
    (liquidity, amount0, amount1, pool) = addLiquidity(
        AddLiquidityParams({
            token0:         params.token0,
            token1:         params.token1,
            fee:            params.fee,
            recipient:      address(this),   // NFPM owns the pool position
            tickLower:      params.tickLower,
            tickUpper:      params.tickUpper,
            amount0Desired:  params.amount0Desired,
            amount1Desired:  params.amount1Desired,
            amount0Min:      params.amount0Min,
            amount1Min:      params.amount1Min
        })
    );

    _mint(params.recipient, (tokenId = _nextId++));

    bytes32 positionKey = PositionKey.compute(address(this), params.tickLower, params.tickUpper);
    (, uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128, , ) = pool.positions(positionKey);

    _positions[tokenId] = Position({
        nonce: 0,
        operator: address(0),
        poolId: cachePoolKey(address(pool), PoolAddress.PoolKey({token0: params.token0, token1: params.token1,
fee: params.fee})),
        tickLower: params.tickLower,
        tickUpper: params.tickUpper,
        liquidity: liquidity,
        feeGrowthInside0LastX128: feeGrowthInside0X128,
        feeGrowthInside1LastX128: feeGrowthInside1X128,
        tokensOwed0: 0,
        tokensOwed1: 0
    });

    emit IncreaseLiquidity(tokenId, liquidity, amount0, amount1);
}
```

14.3 Liquidity Math for Optimal Amounts

Given `amount0Desired` and `amount1Desired`, the NFPM computes the optimal liquidity `L` without wasting either token:

```
// LiquidityAmounts.sol
function getLiquidityForAmounts(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint256 amount0,
    uint256 amount1
) internal pure returns (uint128 liquidity) {
    if (sqrtRatioAX96 > sqrtRatioBX96)
        (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    if (sqrtRatioX96 <= sqrtRatioAX96) {
        // Current price below range — all token0
        liquidity = getLiquidityForAmount0(sqrtRatioAX96, sqrtRatioBX96, amount0);
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        // Price in range — limited by the binding token
        uint128 liquidity0 = getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96, amount0);
        uint128 liquidity1 = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96, amount1);
        liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    } else {
        // Current price above range — all token1
        liquidity = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioBX96, amount1);
    }
}

function getLiquidityForAmount0(uint160 sqrtRatioAX96, uint160 sqrtRatioBX96, uint256 amount0)
    internal pure returns (uint128 liquidity)
{
    // L = amount0 * (√p_b * √p_a) / (√p_b - √p_a)
    uint256 intermediate = FullMath.mulDiv(sqrtRatioAX96, sqrtRatioBX96, FixedPoint96.Q96);
    return toUint128(FullMath.mulDiv(amount0, intermediate, sqrtRatioBX96 - sqrtRatioAX96));
}

function getLiquidityForAmount1(uint160 sqrtRatioAX96, uint160 sqrtRatioBX96, uint256 amount1)
    internal pure returns (uint128 liquidity)
{
    // L = amount1 / (√p_b - √p_a)
    return toUint128(FullMath.mulDiv(amount1, FixedPoint96.Q96, sqrtRatioBX96 - sqrtRatioAX96));
}
```

14.4 Callback Pattern (Mint Callback)

NFPM implements `IUniswapV3MintCallback`:

```
function uniswapV3MintCallback(
    uint256 amount0Owed,
    uint256 amount1Owed,
    bytes calldata data
) external override {
    MintCallbackData memory decoded = abi.decode(data, (MintCallbackData));
    CallbackValidation.verifyCallback(factory, decoded.poolKey);  // validate caller is the pool

    if (amount0Owed > 0)
        pay(decoded.poolKey.token0, decoded.payer, msg.sender, amount0Owed);
    if (amount1Owed > 0)
        pay(decoded.poolKey.token1, decoded.payer, msg.sender, amount1Owed);
}
```

---

15. Periphery: SwapRouter & Quoter

15.1 SwapRouter — Path Encoding

The `SwapRouter` handles both single-hop and multi-hop swaps. Multi-hop paths are encoded as a tightly packed `bytes` sequence:

```
path = token0 ++ fee01 ++ token1 ++ fee12 ++ token2 ++ ...
```

Where:

- `token` = 20 bytes (address)
- `fee` = 3 bytes (uint24)

Each hop: `20 + 3 = 23 bytes`. A 3-hop path: `20 + 3 + 20 + 3 + 20 = 66 bytes`.

```
// Path.sol
uint256 private constant ADDR_SIZE = 20;
uint256 private constant FEE_SIZE  = 3;
uint256 private constant NEXT_OFFSET  = ADDR_SIZE + FEE_SIZE;
uint256 private constant POP_OFFSET   = NEXT_OFFSET + ADDR_SIZE;
uint256 private constant MULTIPLE_POOLS_MIN_LENGTH = POP_OFFSET + NEXT_OFFSET;

function hasMultiplePools(bytes memory path) internal pure returns (bool) {
    return path.length >= MULTIPLE_POOLS_MIN_LENGTH;
}

function decodeFirstPool(bytes memory path)
    internal pure returns (address tokenA, address tokenB, uint24 fee)
{
    tokenA = path.toAddress(0);
    fee    = path.toUint24(ADDR_SIZE);
    tokenB = path.toAddress(NEXT_OFFSET);
}

function skipToken(bytes memory path) internal pure returns (bytes memory) {
    return path.slice(NEXT_OFFSET, path.length - NEXT_OFFSET);
}
```

15.2 Exact Input Swap

```
// SwapRouter.sol
function exactInput(ExactInputParams memory params)
    external payable override checkDeadline(params.deadline)
    returns (uint256 amountOut)
{
    address payer = msg.sender;  // transfers from caller on first hop

    while (true) {
        bool hasMultiplePools = params.path.hasMultiplePools();

        // Execute one hop
        params.amountIn = exactInputInternal(
            params.amountIn,
            hasMultiplePools ? address(this) : params.recipient,  // intermediate hops → router
            0,                                                     // no price limit
            SwapCallbackData({
                path:  params.path.getFirstPool(),
                payer: payer
            })
        );

        if (hasMultiplePools) {
            payer = address(this);   // subsequent hops paid by router from previous output
            params.path = params.path.skipToken();
        } else {
            amountOut = params.amountIn;
            break;
        }
    }

    require(amountOut >= params.amountOutMinimum, 'Too little received');
}
```

15.3 SwapRouter Callback

```
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata _data
) external override {
    require(amount0Delta > 0 || amount1Delta > 0);
    SwapCallbackData memory data = abi.decode(_data, (SwapCallbackData));
    (address tokenIn, address tokenOut, uint24 fee) = data.path.decodeFirstPool();
    CallbackValidation.verifyCallback(factory, tokenIn, tokenOut, fee);

    (bool isExactInput, uint256 amountToPay) = amount0Delta > 0
        ? (tokenIn < tokenOut, uint256(amount0Delta))
        : (tokenOut < tokenIn, uint256(amount1Delta));

    if (isExactInput) {
        pay(tokenIn, data.payer, msg.sender, amountToPay);
    } else {
        // Exact output — multi-hop path is encoded in reverse; decode next pool
        if (data.path.hasMultiplePools()) {
            data.path = data.path.skipToken();
            exactOutputInternal(amountToPay, msg.sender, 0, data);
        } else {
            amountInCached = amountToPay;
            pay(tokenIn, data.payer, msg.sender, amountToPay);
        }
    }
}
```

15.4 Quoter — Off-Chain Simulation

The `Quoter` contract simulates swaps without executing them by using `try/catch` with a deliberate revert-to-return pattern:

```
// Quoter.sol
function quoteExactInputSingle(
    address tokenIn,
    address tokenOut,
    uint24  fee,
    uint256 amountIn,
    uint160 sqrtPriceLimitX96
) public override returns (uint256 amountOut) {
    bool zeroForOne = tokenIn < tokenOut;

    try pool.swap(
        address(this),  // recipient (never actually receives)
        zeroForOne,
        amountIn.toInt256(),
        sqrtPriceLimitX96 == 0
            ? (zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.MAX_SQRT_RATIO - 1)
            : sqrtPriceLimitX96,
        abi.encodePacked(tokenIn, fee, tokenOut)
    ) {} catch (bytes memory reason) {
        return parseRevertReason(reason);
    }
}

// The swap callback reverts with the result encoded as bytes
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes memory path
) external view override {
    address tokenIn  = path.toAddress(0);
    address tokenOut = path.toAddress(Path.NEXT_OFFSET);
    (address tokenA,) = tokenIn < tokenOut ? (tokenIn, tokenOut) : (tokenOut, tokenIn);

    uint256 amountToPay = amount0Delta > 0 ? uint256(amount0Delta) : uint256(amount1Delta);
    uint256 amountReceived = amount0Delta > 0 ? uint256(-amount1Delta) : uint256(-amount0Delta);

    // Revert with the result — caught above by try/catch
    assembly {
        let ptr := mload(0x40)
        mstore(ptr, amountReceived)
        revert(ptr, 32)
    }
}
```

The `Quoter` is **not gas-efficient** and should only be called off-chain (`eth_call`). For on-chain quotes, use the `QuoterV2` which also returns tick and price data.

---

16. Protocol Constants & Bounds

16.1 Core Protocol Constants

Constant	Value	Description
`MIN_TICK`	`-887272`	Minimum tick (≈ price 5.6×10^{-45})
`MAX_TICK`	`887272`	Maximum tick (≈ price 1.8×10^{44})
`MIN_SQRT_RATIO`	`4295128739`	`getSqrtRatioAtTick(MIN_TICK)`
`MAX_SQRT_RATIO`	`1461446703485210103287273052203988822378723970342`	`getSqrtRatioAtTick(MAX_TICK)`
`Q96`	`2^96 = 79228162514264337593543950336`	Fixed-point scaling for sqrtPriceX96
`Q128`	`2^128`	Fixed-point scaling for fee growth accumulators

Constant	Value	Description
`MAX_UINT128`	`2^128 - 1`	Max liquidity, max fees owed
`POOL_INIT_CODE_HASH`	`0xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54`	For deterministic pool address computation

16.2 Fee Tier Constants

Constant	Value	Meaning
Fee denominator	`1,000,000` (1e6)	Fee is expressed in parts per million
Fee 100	`100` ppm	0.01%
Fee 500	`500` ppm	0.05%
Fee 3000	`3000` ppm	0.30%
Fee 10000	`10000` ppm	1.00%

16.3 Per-Tick Liquidity Cap

Each tick has a maximum liquidity cap to prevent a single position from monopolizing a tick and causing integer overflow in `liquidityGross`:

```
// Tick.sol
function tickSpacingToMaxLiquidityPerTick(int24 tickSpacing)
    internal pure returns (uint128)
{
    int24 minTick = (TickMath.MIN_TICK / tickSpacing) * tickSpacing;
    int24 maxTick = (TickMath.MAX_TICK / tickSpacing) * tickSpacing;
    uint24 numTicks = uint24((maxTick - minTick) / tickSpacing) + 1;
    return type(uint128).max / numTicks;
}
```

For tickSpacing=60 (fee 3000): `maxLiquidityPerTick ≈ 1.1 × 10³⁴`.

16.4 Oracle Cardinality Limits

Parameter	Value	Notes
Max cardinality	65,535	`uint16` max, ring buffer size
Default cardinality	1	Single slot at initialization
Gas to initialize slot	~20,000	Cold SSTORE (first-time)
Gas to expand cardinality	`(N - old) × 20,000`	One SSTORE per new slot
Gas to write observation	~5,000	Warm SSTORE

17. Notable Design Decisions

17.1 Callback Architecture (Pull-Over-Push)

Every state-changing operation in v3 (`swap`, `mint`, `flash`) uses the **optimistic transfer + callback** pattern:

1. Pool performs the state update and transfers output tokens **first**
2. Pool invokes a callback on `msg.sender`
3. Inside the callback, the caller is expected to provide the input tokens
4. Pool verifies its balance increased by at least the required amount

```
Pool → transfer output → callback(caller) → caller pays input → Pool checks balance
```

This pattern eliminates the need for `approve()` + `transferFrom()` sequences and enables atomic multi-action compositions (e.g., flash loans, arbitrage, just-in-time liquidity).

17.2 Reentrancy Guard via `slot0.unlocked`

V3's reentrancy guard repurposes the `unlocked` boolean in `slot0` (packed storage):

```
modifier lock() {
    require(slot0.unlocked, 'LOK');
    slot0.unlocked = false;
    _;
    slot0.unlocked = true;
}
```

Reading `slot0` at the beginning of `swap()` and locking early ensures that the reentrancy guard itself is gas-efficient (a single warm SLOAD of the already-hot slot0 slot).

17.3 `noDelegateCall` Guard

The `noDelegateCall` modifier prevents pools from being used as delegate call targets, which would allow the caller to manipulate pool storage through the pool's code:

```
// NoDelegateCall.sol
abstract contract NoDelegateCall {
    address private immutable original;

    constructor() { original = address(this); }

    modifier noDelegateCall() {
        require(address(this) == original);
        _;
    }
}
```

17.4 Singleton-Style Position Key

Position state is stored in a flat mapping indexed by `keccak256(owner ++ tickLower ++ tickUpper)`. This avoids nested mappings, reduces storage pointer levels, and makes position lookups a single SLOAD:

```
// Position.sol
function get(
    mapping(bytes32 => Position.Info) storage self,
    address owner,
    int24   tickLower,
    int24   tickUpper
) internal view returns (Position.Info storage position) {
    position = self[keccak256(abi.encodePacked(owner, tickLower, tickUpper))];
}
```

17.5 Geometric vs Arithmetic TWAP

V3's TWAP accumulates `tick × time` (the log of price), so the computed TWAP is a **geometric mean price** (recovered via `1.0001^avgTick`). Geometric TWAPs are more resistant to manipulation than arithmetic TWAPs because large but brief price distortions have less effect on the geometric average.

V2 accumulated price itself (`price × time`), yielding an arithmetic mean — slightly easier to manipulate with large flash-loan-sized distortions.

17.6 Just-In-Time (JIT) Liquidity

Concentrated liquidity enables a strategy where a sophisticated actor adds a large position in the exact tick range of a pending swap (in the same block), earns the full swap fee, then removes it immediately:

```
block N:
  tx 1: mint(wide range, small L)  [JIT sandwich start]
```

```
    tx 2: swap by victim             [JIT earns fee]
    tx 3: burn + collect             [JIT sandwich end]
```

This is a pure MEV vector — the JIT provider earns fees without sustained commitment. V4 partially addresses this via hook-level mechanisms.

17.7 Price Initialization and the `initialize()` Function

Unlike v2 pairs which set the initial price on the first `mint()`, v3 pools require explicit initialization:

```
function initialize(uint160 sqrtPriceX96) external override {
    require(slot0.sqrtPriceX96 == 0, 'AI');  // already initialized

    int24 tick = TickMath.getTickAtSqrtRatio(sqrtPriceX96);

    (uint16 cardinality, uint16 cardinalityNext) = observations.initialize(_blockTimestamp());

    slot0 = Slot0({
        sqrtPriceX96:               sqrtPriceX96,
        tick:                        tick,
        observationIndex:            0,
        observationCardinality:      cardinality,
        observationCardinalityNext:  cardinalityNext,
        feeProtocol:                 0,
        unlocked:                    true
    });

    emit Initialize(sqrtPriceX96, tick);
}
```

The pool is unusable until `initialize()` is called. This separation allows the pool contract to be deployed (for address pre-computation) before setting the initial price.

17.8 Token Ordering and Direction Convention

Token ordering is always deterministic: `token0 < token1` (by address value). All pool math and swap directions are defined relative to this ordering:

- `zeroForOne = true`: swapping token0 for token1 → price goes down (`sqrtPriceX96` decreases)
- `zeroForOne = false`: swapping token1 for token0 → price goes up (`sqrtPriceX96` increases)

Signed `int256 amount` convention in `swap()`:

- `amountSpecified > 0`: exact input
- `amountSpecified < 0`: exact output

Returned values:

- `amount0 > 0`: pool receives token0 (user pays)
- `amount0 < 0`: pool sends token0 (user receives)
- Same for `amount1`
