Module 4: Block Explorer Analytics

Course: Tool Usage Tutorials
Module: 4 of 6
Duration: 85 minutes
Level: Intermediate
Author: Obelisk Core

Table of Contents

Introduction to Block Explorer Analytics

Block explorers are essential tools for MEV practitioners, providing deep insights into on-chain activity, transaction patterns, and profit opportunities. Understanding how to effectively analyze blockchain data is crucial for identifying MEV opportunities and optimizing trading strategies.

Why Block Explorer Analytics Matter for MEV

Opportunity Identification
- Detect profitable arbitrage opportunities across DEXs
- Identify liquidation opportunities in lending protocols
- Spot sandwich attack targets and front-running opportunities
- Monitor large trades that create MEV opportunities

Competition Analysis
- Analyze successful MEV transactions by other searchers
- Understand gas bidding patterns and strategies

- Study transaction timing and ordering techniques
- Identify emerging MEV strategies and trends

Strategy Optimization
- Measure the effectiveness of your MEV strategies
- Calculate actual vs. theoretical profits
- Optimize gas pricing and transaction timing
- Identify reasons for failed transactions

Risk Management
- Monitor for unexpected protocol changes
- Track large movements that could affect markets
- Identify potential smart contract vulnerabilities
- Monitor for unusual transaction patterns

Key Analytics Platforms

1. **Etherscan**: Primary Ethereum block explorer with comprehensive APIs
2. **Dune Analytics**: Powerful SQL-based blockchain data analytics platform
3. **The Graph**: Decentralized indexing protocol for blockchain data
4. **Nansen**: Advanced on-chain analytics with labeled addresses
5. **Chainlink**: Blockchain data feeds and analytics
6. **DefiLlama**: DeFi protocol analytics and TVL tracking

Etherscan API Integration

Setting Up Etherscan API

```
import requests
import time
import json
from typing import Dict, List, Any, Optional
from datetime import datetime, timedelta
import pandas as pd

class EtherscanAPI:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.etherscan.io/api"
        self.rate_limit_delay = 0.2  # 5 requests per second

    def _make_request(self, params: Dict[str, Any]) -> Dict[str, Any]:
        """Make rate-limited API request to Etherscan"""

        params['apikey'] = self.api_key

        try:
            response = requests.get(self.base_url, params=params,
timeout=10)
            response.raise_for_status()

            time.sleep(self.rate_limit_delay)  # Rate limiting

            return response.json()

        except Exception as e:
            print(f"Etherscan API error: {e}")
            return {"status": "0", "message": str(e)}

    def get_latest_block(self) -> Optional[int]:
        """Get the latest block number"""

        params = {
            'module': 'proxy',
            'action': 'eth_blockNumber'
        }

        result = self._make_request(params)

        if result.get('result'):
```

```
            return int(result['result'], 16)
        return None

    def get_block_transactions(self, block_number: int) ->
List[Dict[str, Any]]:
        """Get all transactions in a specific block"""

        params = {
            'module': 'proxy',
            'action': 'eth_getBlockByNumber',
            'tag': hex(block_number),
            'boolean': 'true'
        }

        result = self._make_request(params)

        if result.get('result') and
result['result'].get('transactions'):
            return result['result']['transactions']
        return []

    def get_transaction_receipt(self, tx_hash: str) ->
Optional[Dict[str, Any]]:
        """Get transaction receipt with logs"""

        params = {
            'module': 'proxy',
            'action': 'eth_getTransactionReceipt',
            'txhash': tx_hash
        }

        result = self._make_request(params)

        if result.get('result'):
            return result['result']
        return None

    def get_account_transactions(self, address: str, start_block: int =
0,
                                 end_block: int = 99999999) ->
List[Dict[str, Any]]:
        """Get transactions for a specific account"""
```

```
        params = {
            'module': 'account',
            'action': 'txlist',
            'address': address,
            'startblock': start_block,
            'endblock': end_block,
            'page': 1,
            'offset': 1000,
            'sort': 'desc'
        }

        result = self._make_request(params)

        if result.get('result'):
            return result['result']
        return []

    def get_erc20_transfers(self, contract_address: str = None,
                            address: str = None) -> List[Dict[str,
Any]]:
        """Get ERC20 token transfers"""

        params = {
            'module': 'account',
            'action': 'tokentx',
            'page': 1,
            'offset': 1000,
            'sort': 'desc'
        }

        if contract_address:
            params['contractaddress'] = contract_address
        if address:
            params['address'] = address

        result = self._make_request(params)

        if result.get('result'):
            return result['result']
        return []
```

```
    def get_gas_tracker(self) -> Dict[str, Any]:
        """Get current gas price recommendations"""

        params = {
            'module': 'gastracker',
            'action': 'gasoracle'
        }

        result = self._make_request(params)

        if result.get('result'):
            return result['result']
        return {}

# Initialize Etherscan API
etherscan = EtherscanAPI("YOUR_API_KEY_HERE")
```

MEV Transaction Analysis

```
class MEVTransactionAnalyzer:
    def __init__(self, etherscan_api: EtherscanAPI):
        self.etherscan = etherscan_api

        # Known MEV contract addresses
        self.mev_contracts = {
            'flashbots_bundle_executor': '0x...',
            'mev_boost_relay': '0x...',
            'arbitrage_bots': [
                '0x...',  # Known arbitrage bot addresses
                '0x...'
            ],
            'sandwich_attackers': [
                '0x...',  # Known sandwich attack addresses
                '0x...'
            ]
        }

        # DEX contract addresses for analysis
        self.dex_contracts = {
            'uniswap_v2_router':
'0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
            'uniswap_v3_router':
'0xE592427A0AEce92De3Edee1F18E0157C05861564',
            'sushiswap_router':
'0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F',
            'balancer_vault':
'0xBA12222222228d8Ba445958a75a0704d566BF2C8',
            'curve_registry':
'0x90E00ACe148ca3b23Ac1bC8C240C2a7Dd9c2d7f5'
        }

    def analyze_block_for_mev(self, block_number: int) -> Dict[str,
Any]:
        """Analyze a block for MEV activity"""

        print(f"Analyzing block {block_number} for MEV activity...")

        # Get all transactions in the block
        transactions =
self.etherscan.get_block_transactions(block_number)
```

```
        if not transactions:
            return {"error": "No transactions found in block"}

        mev_analysis = {
            'block_number': block_number,
            'total_transactions': len(transactions),
            'potential_mev_transactions': [],
            'arbitrage_opportunities': [],
            'sandwich_attacks': [],
            'liquidations': [],
            'gas_analysis': {},
            'timestamp': datetime.utcnow().isoformat()
        }

        # Analyze each transaction
        for i, tx in enumerate(transactions):
            tx_analysis = self.analyze_transaction_for_mev(tx)

            if tx_analysis['is_potential_mev']:

mev_analysis['potential_mev_transactions'].append(tx_analysis)

                # Categorize MEV types
                if tx_analysis['mev_type'] == 'arbitrage':

mev_analysis['arbitrage_opportunities'].append(tx_analysis)
                elif tx_analysis['mev_type'] == 'sandwich':

mev_analysis['sandwich_attacks'].append(tx_analysis)
                elif tx_analysis['mev_type'] == 'liquidation':
                    mev_analysis['liquidations'].append(tx_analysis)

        # Analyze gas patterns
        mev_analysis['gas_analysis'] =
self.analyze_gas_patterns(transactions)

        # Summary statistics
        mev_analysis['summary'] = {
            'mev_transaction_count':
len(mev_analysis['potential_mev_transactions']),
            'mev_percentage':
len(mev_analysis['potential_mev_transactions']) / len(transactions) *
```

```
100,
            'arbitrage_count':
len(mev_analysis['arbitrage_opportunities']),
            'sandwich_count': len(mev_analysis['sandwich_attacks']),
            'liquidation_count': len(mev_analysis['liquidations'])
        }

        return mev_analysis

    def analyze_transaction_for_mev(self, tx: Dict[str, Any]) ->
Dict[str, Any]:
        """Analyze individual transaction for MEV indicators"""

        analysis = {
            'hash': tx['hash'],
            'from': tx['from'],
            'to': tx['to'],
            'value': int(tx['value'], 16) if tx['value'] != '0x' else
0,
            'gas_price': int(tx['gasPrice'], 16),
            'gas_used': int(tx['gas'], 16),
            'is_potential_mev': False,
            'mev_type': None,
            'confidence': 0,
            'indicators': []
        }

        # Check for known MEV contract interactions
        if tx['to'] in self.mev_contracts.get('arbitrage_bots', []):
            analysis['is_potential_mev'] = True
            analysis['mev_type'] = 'arbitrage'
            analysis['confidence'] = 0.9
            analysis['indicators'].append('known_arbitrage_bot')

        # Check for high gas price (potential front-running)
        current_gas = self.etherscan.get_gas_tracker()
        if current_gas:
            fast_gas_price = int(current_gas.get('FastGasPrice', 0)) *
10**9
            if analysis['gas_price'] > fast_gas_price * 2:  # 2x fast
gas price
                analysis['indicators'].append('high_gas_price')
```

```
                analysis['confidence'] += 0.3

        # Check for DEX interactions
        if tx['to'] in self.dex_contracts.values():
            analysis['indicators'].append('dex_interaction')
            analysis['confidence'] += 0.2

        # Check for large value transfers
        if analysis['value'] > 10 * 10**18:  # > 10 ETH
            analysis['indicators'].append('large_value_transfer')
            analysis['confidence'] += 0.1

        # Determine if potentially MEV based on indicators
        if analysis['confidence'] > 0.5:
            analysis['is_potential_mev'] = True

        # Determine MEV type if not already set
        if analysis['is_potential_mev'] and not analysis['mev_type']:
            if 'dex_interaction' in analysis['indicators']:
                analysis['mev_type'] = 'arbitrage'
            elif 'high_gas_price' in analysis['indicators']:
                analysis['mev_type'] = 'front_running'
            else:
                analysis['mev_type'] = 'unknown'

        return analysis

    def analyze_gas_patterns(self, transactions: List[Dict[str, Any]])
-> Dict[str, Any]:
        """Analyze gas price patterns in block transactions"""

        gas_prices = []

        for tx in transactions:
            if tx.get('gasPrice'):
                gas_price = int(tx['gasPrice'], 16)
                gas_prices.append(gas_price)

        if not gas_prices:
            return {}

        gas_prices.sort()
```

```
        return {
            'min_gas_price': min(gas_prices),
            'max_gas_price': max(gas_prices),
            'median_gas_price': gas_prices[len(gas_prices) // 2],
            'avg_gas_price': sum(gas_prices) / len(gas_prices),
            'gas_price_range': max(gas_prices) - min(gas_prices),
            'high_gas_transactions': sum(1 for gp in gas_prices if gp >
gas_prices[-10]),  # Top 10
            'total_transactions': len(gas_prices)
        }

# Example usage
def analyze_recent_mev_activity():
    """Analyze recent MEV activity"""

    analyzer = MEVTransactionAnalyzer(etherscan)

    # Get latest block
    latest_block = etherscan.get_latest_block()
    if not latest_block:
        print("Failed to get latest block")
        return

    print(f"Latest block: {latest_block}")

    # Analyze last 5 blocks
    for i in range(5):
        block_number = latest_block - i
        analysis = analyzer.analyze_block_for_mev(block_number)

        if 'error' not in analysis:
            summary = analysis['summary']
            print(f"\nBlock {block_number}:")
            print(f"  Total transactions:
{summary['mev_transaction_count']}")
            print(f"  MEV percentage: {summary['mev_percentage']:.1f}
%")
            print(f"  Arbitrage: {summary['arbitrage_count']}")
            print(f"  Sandwich: {summary['sandwich_count']}")
            print(f"  Liquidations: {summary['liquidation_count']}")
```

```
# Run analysis
# analyze_recent_mev_activity()
```

DEX Arbitrage Detection

```
class ArbitrageDetector:
    def __init__(self, etherscan_api: EtherscanAPI):
        self.etherscan = etherscan_api

        # Token addresses for analysis
        self.tokens = {
            'WETH': '0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2',
            'USDC': '0xA0b86a33E6417c8f4CF46d764e1c1a0D8e34b4E',
            'USDT': '0xdAC17F958D2ee523a2206206994597C13D831ec7',
            'DAI': '0x6B175474E89094C44Da98b954EedeAC495271d0F',
            'WBTC': '0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599'
        }

        # DEX router addresses for tracking
        self.dex_routers = {
            'uniswap_v2': '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
            'sushiswap': '0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F',
            'pancakeswap': '0x10ED43C718714eb63d5aA57B78B54704E256024E'
        }

    def detect_arbitrage_transactions(self, block_number: int) ->
List[Dict[str, Any]]:
        """Detect potential arbitrage transactions in a block"""

        transactions =
self.etherscan.get_block_transactions(block_number)
        arbitrage_opportunities = []

        # Group transactions by sender to find arbitrage sequences
        sender_transactions = {}

        for tx in transactions:
            sender = tx['from']
            if sender not in sender_transactions:
                sender_transactions[sender] = []
            sender_transactions[sender].append(tx)

        # Analyze each sender's transactions for arbitrage patterns
        for sender, sender_txs in sender_transactions.items():
            if len(sender_txs) >= 2:  # Need at least 2 transactions
for arbitrage
                arbitrage_analysis =
```

```
self.analyze_arbitrage_sequence(sender_txs)

                if arbitrage_analysis['is_arbitrage']:
                    arbitrage_opportunities.append({
                        'sender': sender,
                        'block_number': block_number,
                        'transaction_count': len(sender_txs),
                        'analysis': arbitrage_analysis,
                        'estimated_profit':
arbitrage_analysis.get('estimated_profit', 0)
                    })

        return arbitrage_opportunities

    def analyze_arbitrage_sequence(self, transactions: List[Dict[str,
Any]]) -> Dict[str, Any]:
        """Analyze sequence of transactions for arbitrage patterns"""

        analysis = {
            'is_arbitrage': False,
            'confidence': 0,
            'dex_interactions': [],
            'token_flows': {},
            'estimated_profit': 0,
            'gas_cost': 0
        }

        total_gas_cost = 0
        dex_count = 0

        for tx in transactions:
            # Calculate gas cost
            gas_price = int(tx['gasPrice'], 16)
            gas_used = int(tx['gas'], 16)  # This is gas limit, not
actual used
            total_gas_cost += gas_price * gas_used

            # Check for DEX interactions
            if tx['to'] in self.dex_routers.values():
                dex_count += 1

                # Identify which DEX
```

```
                for dex_name, router_address in
self.dex_routers.items():
                    if tx['to'] == router_address:
                        analysis['dex_interactions'].append({
                            'dex': dex_name,
                            'tx_hash': tx['hash'],
                            'value': int(tx['value'], 16) if
tx['value'] != '0x' else 0
                        })

        analysis['gas_cost'] = total_gas_cost

        # Arbitrage indicators
        if dex_count >= 2:  # Interactions with multiple DEXs
            analysis['confidence'] += 0.7

        if len(set(interaction['dex'] for interaction in
analysis['dex_interactions'])) >= 2:
            analysis['confidence'] += 0.3  # Multiple different DEXs

        # Determine if this looks like arbitrage
        if analysis['confidence'] >= 0.8:
            analysis['is_arbitrage'] = True

        return analysis

    def track_arbitrage_profits(self, arbitrage_tx: Dict[str, Any]) ->
Dict[str, Any]:
        """Track actual profits from arbitrage transaction"""

        sender = arbitrage_tx['sender']
        block_number = arbitrage_tx['block_number']

        # Get ETH balance changes

# This would require getting transaction receipts and analyzing state
changes
        # Simplified implementation

        profit_analysis = {
            'gross_profit': 0,
            'gas_cost': arbitrage_tx['analysis']['gas_cost'],
```

```
            'net_profit': 0,
            'roi': 0,
            'success': True
        }

        # In a real implementation, you would:
        # 1. Get transaction receipts for all transactions
        # 2. Analyze ERC20 transfer events
        # 3. Calculate token balance changes
        # 4. Convert to USD/ETH values
        # 5. Calculate actual profit

        return profit_analysis

# Example arbitrage detection
def scan_for_arbitrage_opportunities():
    """Scan recent blocks for arbitrage opportunities"""

    detector = ArbitrageDetector(etherscan)

    latest_block = etherscan.get_latest_block()
    if not latest_block:
        return

    print("Scanning for arbitrage opportunities...")

    arbitrage_found = []

    # Scan last 10 blocks
    for i in range(10):
        block_number = latest_block - i
        opportunities =
detector.detect_arbitrage_transactions(block_number)

        if opportunities:
            arbitrage_found.extend(opportunities)
            print(f"Block {block_number}: Found {len(opportunities)}
potential arbitrage transactions")

    # Analyze findings
    if arbitrage_found:
        print(f"\nTotal arbitrage opportunities found:
```

```
{len(arbitrage_found)}")

        # Show top opportunities by estimated profit
        top_opportunities = sorted(arbitrage_found,
                                   key=lambda x: x['estimated_profit'],
                                   reverse=True)[:5]

        print("\nTop arbitrage opportunities:")
        for i, opp in enumerate(top_opportunities, 1):
            print(f"{i}. Sender: {opp['sender'][:10]}...")
            print(f"   Block: {opp['block_number']}")
            print(f"   Transactions: {opp['transaction_count']}")
            print(f"   Estimated profit: {opp['estimated_profit'] /
10**18:.6f} ETH")

# Run arbitrage scanning
# scan_for_arbitrage_opportunities()
```

Dune Analytics for MEV Research

Setting Up Dune Analytics

```
import requests
import pandas as pd
from typing import Dict, List, Any, Optional
import time

class DuneAnalytics:
    def __init__(self, api_key: str):
        self.api_key = api_key
        self.base_url = "https://api.dune.com/api/v1"
        self.headers = {
            "X-Dune-API-Key": api_key,
            "Content-Type": "application/json"
        }

    def execute_query(self, query_id: int, parameters: Dict[str, Any] =
None) -> Dict[str, Any]:
        """Execute a Dune query"""

        url = f"{self.base_url}/query/{query_id}/execute"

        payload = {}
        if parameters:
            payload["query_parameters"] = parameters

        response = requests.post(url, headers=self.headers,
json=payload)

        if response.status_code == 200:
            return response.json()
        else:
            return {"error": f"HTTP {response.status_code}:
{response.text}"}

    def get_execution_status(self, execution_id: str) -> Dict[str,
Any]:
        """Get execution status"""

        url = f"{self.base_url}/execution/{execution_id}/status"
        response = requests.get(url, headers=self.headers)

        if response.status_code == 200:
            return response.json()
```

```
        else:
            return {"error": f"HTTP {response.status_code}:
{response.text}"}

    def get_execution_results(self, execution_id: str) -> Dict[str,
Any]:
        """Get execution results"""

        url = f"{self.base_url}/execution/{execution_id}/results"
        response = requests.get(url, headers=self.headers)

        if response.status_code == 200:
            return response.json()
        else:
            return {"error": f"HTTP {response.status_code}:
{response.text}"}

    def run_query_and_wait(self, query_id: int, parameters: Dict[str,
Any] = None,
                           max_wait_seconds: int = 300) ->
Optional[pd.DataFrame]:
        """Execute query and wait for results"""

        # Start execution
        execution_result = self.execute_query(query_id, parameters)

        if "error" in execution_result:
            print(f"Failed to start query execution:
{execution_result['error']}")
            return None

        execution_id = execution_result["execution_id"]
        print(f"Started query execution: {execution_id}")

        # Wait for completion
        start_time = time.time()

        while time.time() - start_time < max_wait_seconds:
            status = self.get_execution_status(execution_id)

            if "error" in status:
                print(f"Error checking status: {status['error']}")
```

```
                return None

            state = status.get("state")

            if state == "QUERY_STATE_COMPLETED":
                print("Query completed successfully")
                break
            elif state == "QUERY_STATE_FAILED":
                print("Query execution failed")
                return None
            else:
                print(f"Query state: {state}, waiting...")
                time.sleep(5)
        else:
            print("Query execution timed out")
            return None

        # Get results
        results = self.get_execution_results(execution_id)

        if "error" in results:
            print(f"Error getting results: {results['error']}")
            return None

        # Convert to DataFrame
        if results.get("result", {}).get("rows"):
            df = pd.DataFrame(results["result"]["rows"])
            return df
        else:
            print("No data returned")
            return None

# Initialize Dune Analytics
dune = DuneAnalytics("YOUR_DUNE_API_KEY")
```

MEV Analytics Queries

```
-- Dune Query: Daily MEV Extraction Volume
WITH mev_transactions AS (
    SELECT
        DATE_TRUNC('day', block_time) as date,
        tx_hash,
        "from" as searcher,
        gas_price,
        gas_used,
        (gas_price * gas_used) / 1e18 as gas_cost_eth,
        value / 1e18 as value_eth
    FROM ethereum.transactions
    WHERE block_time >= NOW() - INTERVAL '30 days'
        AND (
            "to" IN (
                '0x...', -- Known MEV contract addresses
                '0x...'
            )
            OR gas_price > 100e9  -- High gas price transactions
        )
)

SELECT
    date,
    COUNT(*) as mev_transaction_count,
    COUNT(DISTINCT searcher) as unique_searchers,
    SUM(gas_cost_eth) as total_gas_cost_eth,
    AVG(gas_cost_eth) as avg_gas_cost_eth,
    SUM(value_eth) as total_value_eth
FROM mev_transactions
GROUP BY date
ORDER BY date DESC;
```

```
-- Dune Query: Top MEV Searchers by Profit
WITH searcher_stats AS (
    SELECT
        "from" as searcher_address,
        COUNT(*) as transaction_count,
        SUM((gas_price * gas_used) / 1e18) as total_gas_cost,
        AVG((gas_price * gas_used) / 1e18) as avg_gas_cost,
        MAX(block_time) as last_activity
    FROM ethereum.transactions
    WHERE block_time >= NOW() - INTERVAL '7 days'
        AND (
            "to" IN (
                '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D', --
Uniswap V2
                '0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F'  --
SushiSwap
            )
            OR gas_price > 50e9
        )
    GROUP BY "from"
    HAVING COUNT(*) >= 10  -- Active searchers only
)

SELECT
    searcher_address,
    transaction_count,
    total_gas_cost,
    avg_gas_cost,
    last_activity,
    RANK() OVER (ORDER BY total_gas_cost DESC) as gas_rank
FROM searcher_stats
ORDER BY total_gas_cost DESC
LIMIT 50;
```

```
def analyze_mev_trends_with_dune():
    """Analyze MEV trends using Dune Analytics"""

    # Query IDs for pre-created Dune queries
    DAILY_MEV_VOLUME_QUERY = 1234567  # Replace with actual query ID
    TOP_SEARCHERS_QUERY = 1234568     # Replace with actual query ID

    print("Fetching MEV trends from Dune Analytics...")

    # Get daily MEV volume data
    daily_volume_df = dune.run_query_and_wait(DAILY_MEV_VOLUME_QUERY)

    if daily_volume_df is not None:
        print("\nDaily MEV Volume (Last 30 Days):")
        print(daily_volume_df.head(10))

        # Calculate trends
        total_transactions =
daily_volume_df['mev_transaction_count'].sum()
        total_gas_cost = daily_volume_df['total_gas_cost_eth'].sum()
        avg_daily_volume =
daily_volume_df['mev_transaction_count'].mean()

        print(f"\nSummary:")
        print(f"Total MEV transactions: {total_transactions:,}")
        print(f"Total gas cost: {total_gas_cost:.2f} ETH")
        print(f"Average daily MEV transactions: {avg_daily_volume:.
0f}")

    # Get top searchers data
    print("\nFetching top MEV searchers...")
    top_searchers_df = dune.run_query_and_wait(TOP_SEARCHERS_QUERY)

    if top_searchers_df is not None:
        print("\nTop MEV Searchers (Last 7 Days):")
        print(top_searchers_df.head(10))

    return {
        'daily_volume': daily_volume_df,
        'top_searchers': top_searchers_df
    }
```

```
# Create custom MEV analysis queries
class MEVDuneQueries:
    def __init__(self, dune_client: DuneAnalytics):
        self.dune = dune_client

    def get_arbitrage_opportunities_query(self) -> str:
        """Query to find arbitrage opportunities"""

        return """
        WITH dex_swaps AS (
            SELECT
                block_time,
                block_number,
                tx_hash,
                project,
                token_a_symbol,
                token_b_symbol,
                token_a_amount,
                token_b_amount,
                amount_usd
            FROM dex.trades
            WHERE block_time >= NOW() - INTERVAL '1 hour'
                AND project IN ('Uniswap', 'SushiSwap', 'Balancer',
'Curve')
                AND amount_usd > 1000  -- Significant trades only
        ),

        price_differences AS (
            SELECT
                s1.block_number,
                s1.token_a_symbol,
                s1.token_b_symbol,
                s1.project as dex1,
                s2.project as dex2,
                s1.token_b_amount / s1.token_a_amount as price1,
                s2.token_b_amount / s2.token_a_amount as price2,
                ABS(
                    (s1.token_b_amount / s1.token_a_amount) -
                    (s2.token_b_amount / s2.token_a_amount)
                ) / (s1.token_b_amount / s1.token_a_amount) as
price_diff_pct
            FROM dex_swaps s1
```

```
            JOIN dex_swaps s2 ON s1.block_number = s2.block_number
                AND s1.token_a_symbol = s2.token_a_symbol
                AND s1.token_b_symbol = s2.token_b_symbol
                AND s1.project != s2.project
        )

        SELECT
            block_number,
            token_a_symbol,
            token_b_symbol,
            dex1,
            dex2,
            price1,
            price2,
            price_diff_pct,
            CASE
                WHEN price_diff_pct > 0.01 THEN 'High'
                WHEN price_diff_pct > 0.005 THEN 'Medium'
                ELSE 'Low'
            END as opportunity_level
        FROM price_differences
        WHERE price_diff_pct > 0.005  -- At least 0.5% price difference
        ORDER BY price_diff_pct DESC
        LIMIT 100;
        """

    def get_liquidation_opportunities_query(self) -> str:
        """Query to find liquidation opportunities"""

        return """
        WITH aave_positions AS (
            SELECT
                user,
                reserve,
                current_a_token_balance,
                current_stable_debt,
                current_variable_debt,
                liquidation_threshold
            FROM aave_v2.user_reserves
            WHERE current_a_token_balance > 0
                OR current_stable_debt > 0
                OR current_variable_debt > 0
```

```
        ),

        at_risk_positions AS (
            SELECT
                user,
                reserve,
                current_a_token_balance,
                (current_stable_debt + current_variable_debt) as
total_debt,
                CASE
                    WHEN current_a_token_balance > 0
                    THEN (current_stable_debt +
current_variable_debt) / current_a_token_balance
                    ELSE 0
                END as debt_ratio
            FROM aave_positions
        )

        SELECT
            user,
            reserve,
            current_a_token_balance,
            total_debt,
            debt_ratio,
            CASE
                WHEN debt_ratio > 0.95 THEN 'Critical'
                WHEN debt_ratio > 0.90 THEN 'High Risk'
                WHEN debt_ratio > 0.85 THEN 'Medium Risk'
                ELSE 'Low Risk'
            END as liquidation_risk
        FROM at_risk_positions
        WHERE debt_ratio > 0.85  -- At risk positions only
        ORDER BY debt_ratio DESC
        LIMIT 100;
        """

# Run MEV analysis
# mev_data = analyze_mev_trends_with_dune()
```

Advanced On-Chain Analysis

Transaction Flow Analysis

```
class TransactionFlowAnalyzer:
    def __init__(self, etherscan_api: EtherscanAPI):
        self.etherscan = etherscan_api

    def trace_transaction_flow(self, tx_hash: str) -> Dict[str, Any]:
        """Trace the flow of funds in a transaction"""

        # Get transaction receipt with logs
        receipt = self.etherscan.get_transaction_receipt(tx_hash)

        if not receipt:
            return {"error": "Transaction not found"}

        flow_analysis = {
            'transaction_hash': tx_hash,
            'status': receipt.get('status'),
            'gas_used': int(receipt.get('gasUsed', '0'), 16),
            'logs_count': len(receipt.get('logs', [])),
            'token_transfers': [],
            'eth_transfers': [],
            'contract_interactions': [],
            'mev_indicators': []
        }

        # Analyze logs for token transfers and other events
        for log in receipt.get('logs', []):
            log_analysis = self.analyze_log_entry(log)

            if log_analysis['type'] == 'token_transfer':
                flow_analysis['token_transfers'].append(log_analysis)
            elif log_analysis['type'] == 'contract_interaction':

flow_analysis['contract_interactions'].append(log_analysis)

        # Detect MEV patterns
        flow_analysis['mev_indicators'] =
self.detect_mev_patterns(flow_analysis)

        return flow_analysis

    def analyze_log_entry(self, log: Dict[str, Any]) -> Dict[str, Any]:
        """Analyze individual log entry"""
```

```
        log_analysis = {
            'address': log.get('address'),
            'topics': log.get('topics', []),
            'data': log.get('data'),
            'type': 'unknown'
        }

        # Check for ERC20 Transfer event
        if (len(log_analysis['topics']) >= 3 and
            log_analysis['topics'][0] ==
'0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef'):

            log_analysis['type'] = 'token_transfer'
            log_analysis['from'] = '0x' + log_analysis['topics'][1]
[-40:]
            log_analysis['to'] = '0x' + log_analysis['topics'][2][-40:]

            # Decode amount from data field (simplified)
            if log_analysis['data'] and log_analysis['data'] != '0x':
                try:
                    amount_hex = log_analysis['data'][2:]  # Remove 0x
prefix
                    log_analysis['amount'] = int(amount_hex, 16)
                except:
                    log_analysis['amount'] = 0

        # Check for Uniswap Swap event
        elif (len(log_analysis['topics']) >= 1 and
              log_analysis['topics'][0] ==
'0xd78ad95fa46c994b6551d0da85fc275fe613ce37657fb8d5e3d130840159d822'):

            log_analysis['type'] = 'uniswap_swap'
            # Additional parsing would go here

        return log_analysis

    def detect_mev_patterns(self, flow_analysis: Dict[str, Any]) ->
List[str]:
        """Detect MEV patterns in transaction flow"""

        indicators = []
```

```
        # Check for multiple token transfers (potential arbitrage)
        if len(flow_analysis['token_transfers']) >= 4:
            indicators.append('multiple_token_transfers')

        # Check for flash loan patterns
        token_addresses = set()
        for transfer in flow_analysis['token_transfers']:
            token_addresses.add(transfer.get('address'))

        if len(token_addresses) >= 3:  # Multiple different tokens
            indicators.append('multi_token_arbitrage')

        # Check for high gas usage (complex operations)
        if flow_analysis['gas_used'] > 500000:
            indicators.append('high_gas_usage')

        # Check for contract interactions with known DEXs
        dex_interactions = 0
        for interaction in flow_analysis['contract_interactions']:
            if interaction.get('address') in [
                '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',  #
Uniswap V2
                '0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F'   #
SushiSwap
            ]:
                dex_interactions += 1

        if dex_interactions >= 2:
            indicators.append('multiple_dex_interactions')

        return indicators

# MEV Pattern Recognition
class MEVPatternRecognizer:
    def __init__(self):
        self.known_patterns = {
            'sandwich_attack': {
                'description': 'Front-run and back-run a large trade',
                'indicators': ['high_gas_price', 'dex_interaction',
'timing_pattern'],
                'min_confidence': 0.8
```

```
            },
            'arbitrage': {
                'description': 'Price differences across multiple
DEXs',
                'indicators': ['multiple_dex_interactions',
'multi_token_arbitrage'],
                'min_confidence': 0.7
            },
            'liquidation': {
                'description': 'Liquidating undercollateralized
positions',
                'indicators': ['lending_protocol_interaction',
'collateral_transfer'],
                'min_confidence': 0.6
            },
            'flash_loan_arbitrage': {
                'description': 'Using flash loans for arbitrage',
                'indicators': ['flash_loan_pattern',
'multiple_token_transfers'],
                'min_confidence': 0.8
            }
        }

    def classify_mev_transaction(self, flow_analysis: Dict[str, Any]) -
> Dict[str, Any]:
        """Classify MEV transaction type"""

        classification = {
            'primary_type': 'unknown',
            'confidence': 0,
            'all_patterns': [],
            'indicators_found': flow_analysis.get('mev_indicators', [])
        }

        # Check each known pattern
        for pattern_name, pattern_info in self.known_patterns.items():
            confidence = self.calculate_pattern_confidence(
                flow_analysis.get('mev_indicators', []),
                pattern_info['indicators']
            )

            if confidence >= pattern_info['min_confidence']:
```

```
                classification['all_patterns'].append({
                    'type': pattern_name,
                    'confidence': confidence,
                    'description': pattern_info['description']
                })

        # Select primary type (highest confidence)
        if classification['all_patterns']:
            primary = max(classification['all_patterns'], key=lambda x:
x['confidence'])
            classification['primary_type'] = primary['type']
            classification['confidence'] = primary['confidence']

        return classification

    def calculate_pattern_confidence(self, found_indicators: List[str],
                                     required_indicators: List[str]) ->
float:
        """Calculate confidence score for pattern matching"""

        if not required_indicators:
            return 0

        matches = sum(1 for indicator in required_indicators if
indicator in found_indicators)
        return matches / len(required_indicators)

# Example usage
def analyze_mev_transaction_flow():
    """Analyze MEV transaction flow"""

    analyzer = TransactionFlowAnalyzer(etherscan)
    recognizer = MEVPatternRecognizer()

    # Example transaction hash (replace with actual MEV transaction)
    tx_hash = "0x..."

    print(f"Analyzing transaction flow: {tx_hash}")

    # Trace transaction flow
    flow_analysis = analyzer.trace_transaction_flow(tx_hash)
```

```
    if 'error' not in flow_analysis:
        print(f"\nTransaction Analysis:")
        print(f"Status: {flow_analysis['status']}")
        print(f"Gas used: {flow_analysis['gas_used']:,}")
        print(f"Token transfers:
{len(flow_analysis['token_transfers'])}")
        print(f"Contract interactions:
{len(flow_analysis['contract_interactions'])}")
        print(f"MEV indicators: {flow_analysis['mev_indicators']}")

        # Classify MEV type
        classification =
recognizer.classify_mev_transaction(flow_analysis)

        print(f"\nMEV Classification:")
        print(f"Primary type: {classification['primary_type']}")
        print(f"Confidence: {classification['confidence']:.1%}")

        if classification['all_patterns']:
            print(f"All detected patterns:")
            for pattern in classification['all_patterns']:
                print(f"  - {pattern['type']}: {pattern['confidence']:.
1%}")
    else:
        print(f"Error: {flow_analysis['error']}")

# Run transaction flow analysis
# analyze_mev_transaction_flow()
```

MEV Opportunity Detection

Real-Time Opportunity Scanner

```
import asyncio
import websockets
import json
from typing import Dict, List, Any, Callable
import threading
import queue

class MEVOpportunityScanner:
    def __init__(self, etherscan_api: EtherscanAPI):
        self.etherscan = etherscan_api
        self.is_scanning = False
        self.opportunity_queue = queue.Queue()
        self.subscribers = []

        # Opportunity detection thresholds
        self.thresholds = {
            'min_arbitrage_profit': 0.01,  # ETH
            'min_liquidation_value': 1.0,  # ETH
            'max_gas_price': 300 * 10**9,  # 300 gwei
            'min_trade_size': 10000  # USD
        }

    def add_opportunity_subscriber(self, callback: Callable):
        """Add callback for opportunity notifications"""
        self.subscribers.append(callback)

    def notify_subscribers(self, opportunity: Dict[str, Any]):
        """Notify all subscribers of new opportunity"""
        for callback in self.subscribers:
            try:
                callback(opportunity)
            except Exception as e:
                print(f"Error notifying subscriber: {e}")

    async def scan_mempool_for_opportunities(self):
        """Scan mempool for MEV opportunities (simulated)"""

        # In a real implementation, this would connect to:
        # - Ethereum mempool via WebSocket
        # - MEV relay networks
        # - DEX event streams
```

```
        while self.is_scanning:
            try:
                # Simulate finding opportunities
                opportunity = await self.detect_opportunity()

                if opportunity:
                    self.opportunity_queue.put(opportunity)
                    self.notify_subscribers(opportunity)

                await asyncio.sleep(1)  # Check every second

            except Exception as e:
                print(f"Error scanning mempool: {e}")
                await asyncio.sleep(5)

    async def detect_opportunity(self) -> Optional[Dict[str, Any]]:
        """Detect MEV opportunities from current market data"""

        # Get current block
        latest_block = self.etherscan.get_latest_block()
        if not latest_block:
            return None

        # Simulate opportunity detection
        # In reality, this would analyze:
        # - Pending transactions in mempool
        # - Price differences across DEXs
        # - Liquidation candidates in lending protocols
        # - Large trades that create opportunities

        import random

        if random.random() < 0.1:  # 10% chance of finding opportunity
            opportunity_types = ['arbitrage', 'liquidation',
'sandwich']
            opp_type = random.choice(opportunity_types)

            opportunity = {
                'type': opp_type,
                'timestamp': datetime.utcnow().isoformat(),
                'block_number': latest_block,
                'estimated_profit': random.uniform(0.01, 0.5),  # ETH
```

```
                'gas_cost_estimate': random.uniform(0.005, 0.02),  # 
ETH
                'confidence': random.uniform(0.6, 0.95),
                'time_sensitive': True,
                'expiry': datetime.utcnow().timestamp() + 12  # 12 
seconds
            }

            # Add type-specific data
            if opp_type == 'arbitrage':
                opportunity.update({
                    'token_pair': 'WETH/USDC',
                    'dex1': 'Uniswap V2',
                    'dex2': 'SushiSwap',
                    'price_difference': 0.02,  # 2%
                    'trade_size': 10000  # USD
                })
            elif opp_type == 'liquidation':
                opportunity.update({
                    'protocol': 'Aave V2',
                    'collateral_token': 'WETH',
                    'debt_token': 'USDC',
                    'collateral_value': 5.0,  # ETH
                    'liquidation_bonus': 0.05  # 5%
                })
            elif opp_type == 'sandwich':
                opportunity.update({
                    'target_tx_hash': '0x...',
                    'target_trade_size': 50000,  # USD
                    'slippage_opportunity': 0.015  # 1.5%
                })

            return opportunity

        return None

    def evaluate_opportunity_profitability(self, opportunity: Dict[str, 
Any]) -> Dict[str, Any]:
        """Evaluate if opportunity is profitable"""

        evaluation = {
            'is_profitable': False,
```

```
            'net_profit': 0,
            'roi': 0,
            'risk_level': 'unknown',
            'recommendation': 'skip'
        }

        estimated_profit = opportunity.get('estimated_profit', 0)
        gas_cost = opportunity.get('gas_cost_estimate', 0)
        confidence = opportunity.get('confidence', 0)

        # Calculate net profit
        net_profit = estimated_profit - gas_cost
        evaluation['net_profit'] = net_profit

        # Calculate ROI (simplified)
        if gas_cost > 0:
            evaluation['roi'] = net_profit / gas_cost

        # Determine profitability
        if net_profit > self.thresholds['min_arbitrage_profit'] and
confidence > 0.7:
            evaluation['is_profitable'] = True

            # Determine risk level
            if confidence > 0.9 and net_profit > 0.05:
                evaluation['risk_level'] = 'low'
                evaluation['recommendation'] = 'execute'
            elif confidence > 0.8 and net_profit > 0.02:
                evaluation['risk_level'] = 'medium'
                evaluation['recommendation'] = 'consider'
            else:
                evaluation['risk_level'] = 'high'
                evaluation['recommendation'] = 'monitor'

        return evaluation

    def start_scanning(self):
        """Start opportunity scanning"""
        if not self.is_scanning:
            self.is_scanning = True
            asyncio.create_task(self.scan_mempool_for_opportunities())
            print("MEV opportunity scanning started")
```

```
    def stop_scanning(self):
        """Stop opportunity scanning"""
        self.is_scanning = False
        print("MEV opportunity scanning stopped")

# Opportunity Alert System
class MEVAlertSystem:
    def __init__(self, scanner: MEVOpportunityScanner):
        self.scanner = scanner

self.scanner.add_opportunity_subscriber(self.handle_opportunity)
        self.alert_history = []

    def handle_opportunity(self, opportunity: Dict[str, Any]):
        """Handle new opportunity detection"""

        # Evaluate profitability
        evaluation =
self.scanner.evaluate_opportunity_profitability(opportunity)

        # Create alert
        alert = {
            'timestamp': datetime.utcnow().isoformat(),
            'opportunity': opportunity,
            'evaluation': evaluation,
            'alert_level': self.determine_alert_level(evaluation)
        }

        # Store alert
        self.alert_history.append(alert)

        # Send notifications based on alert level
        if alert['alert_level'] in ['high', 'critical']:
            self.send_immediate_alert(alert)

        # Log opportunity
        self.log_opportunity(alert)

    def determine_alert_level(self, evaluation: Dict[str, Any]) -> str:
        """Determine alert priority level"""
```

```
        if not evaluation['is_profitable']:
            return 'info'

        net_profit = evaluation['net_profit']
        risk_level = evaluation['risk_level']

        if net_profit > 0.1 and risk_level == 'low':  # High profit,
low risk
            return 'critical'
        elif net_profit > 0.05 and risk_level in ['low', 'medium']:
            return 'high'
        elif net_profit > 0.02:
            return 'medium'
        else:
            return 'low'

    def send_immediate_alert(self, alert: Dict[str, Any]):
        """Send immediate alert for high-priority opportunities"""

        opportunity = alert['opportunity']
        evaluation = alert['evaluation']

        message = f"""
🚨 MEV OPPORTUNITY ALERT 🚨

Type: {opportunity['type'].upper()}
Estimated Profit: {opportunity['estimated_profit']:.4f} ETH
Net Profit: {evaluation['net_profit']:.4f} ETH
ROI: {evaluation['roi']:.1%}
Confidence: {opportunity['confidence']:.1%}
Risk Level: {evaluation['risk_level']}
Recommendation: {evaluation['recommendation'].upper()}

Time Sensitive: Expires in {opportunity.get('expiry', 0) -
datetime.utcnow().timestamp():.0f} seconds

Details: {json.dumps(opportunity, indent=2)}
        """

        print(message)

        # In production, send to:
```

```
        # - Discord/Telegram webhook
        # - Email alerts
        # - Mobile push notifications
        # - Trading bot systems

    def log_opportunity(self, alert: Dict[str, Any]):
        """Log opportunity for analysis"""

        log_entry = {
            'timestamp': alert['timestamp'],
            'type': alert['opportunity']['type'],
            'profit': alert['evaluation']['net_profit'],
            'confidence': alert['opportunity']['confidence'],
            'alert_level': alert['alert_level'],
            'recommendation': alert['evaluation']['recommendation']
        }

        # Log to file or database
        print(f"OPPORTUNITY LOG: {json.dumps(log_entry)}")

    def get_opportunity_statistics(self, hours: int = 24) -> Dict[str,
Any]:
        """Get opportunity statistics for specified period"""

        cutoff_time = datetime.utcnow() - timedelta(hours=hours)

        recent_alerts = [
            alert for alert in self.alert_history
            if datetime.fromisoformat(alert['timestamp']) > cutoff_time
        ]

        if not recent_alerts:
            return {"message": "No opportunities found in specified
period"}

        # Calculate statistics
        total_opportunities = len(recent_alerts)
        profitable_opportunities = sum(
            1 for alert in recent_alerts
            if alert['evaluation']['is_profitable']
        )
```

```
        total_potential_profit = sum(
            alert['evaluation']['net_profit'] for alert in
recent_alerts
            if alert['evaluation']['is_profitable']
        )

        opportunity_types = {}
        for alert in recent_alerts:
            opp_type = alert['opportunity']['type']
            opportunity_types[opp_type] =
opportunity_types.get(opp_type, 0) + 1

        statistics = {
            'period_hours': hours,
            'total_opportunities': total_opportunities,
            'profitable_opportunities': profitable_opportunities,
            'profitability_rate': profitable_opportunities /
total_opportunities if total_opportunities > 0 else 0,
            'total_potential_profit_eth': total_potential_profit,
            'avg_profit_per_opportunity': total_potential_profit /
profitable_opportunities if profitable_opportunities > 0 else 0,
            'opportunity_types': opportunity_types,
            'alert_levels': {
                level: sum(1 for alert in recent_alerts if
alert['alert_level'] == level)
                for level in ['info', 'low', 'medium', 'high',
'critical']
            }
        }

        return statistics

# Example usage
async def run_mev_opportunity_detection():
    """Run MEV opportunity detection system"""

    # Initialize scanner
    scanner = MEVOpportunityScanner(etherscan)
    alert_system = MEVAlertSystem(scanner)

    print("Starting MEV opportunity detection system...")
```

```
    # Start scanning
    scanner.start_scanning()

    try:
        # Run for a specified time or indefinitely
        await asyncio.sleep(300)  # Run for 5 minutes

    except KeyboardInterrupt:
        print("Stopping MEV opportunity detection...")
    finally:
        scanner.stop_scanning()

    # Print statistics
    stats = alert_system.get_opportunity_statistics(24)
    print(f"\n24-Hour Opportunity Statistics:")
    print(json.dumps(stats, indent=2))

# Run opportunity detection
# asyncio.run(run_mev_opportunity_detection())
```

Summary

In this comprehensive module, you've learned how to:

✅ **Integrate with Etherscan API** for detailed blockchain data analysis and transaction monitoring

✅ **Use Dune Analytics** for powerful SQL-based blockchain research and MEV trend analysis

✅ **Perform advanced on-chain analysis** including transaction flow tracing and pattern recognition

✅ **Detect MEV opportunities** through automated scanning and real-time monitoring systems

✅ **Analyze transaction patterns** to identify arbitrage, liquidation, and sandwich attack opportunities

✅ **Set up monitoring systems** with automated alerts and notification mechanisms

✅ **Optimize data collection** with efficient API usage and rate limiting strategies

✅ **Visualize MEV trends** and create comprehensive analytics dashboards

Block explorer analytics are fundamental to successful MEV operations, providing the data insights needed to identify opportunities, analyze competition, and optimize strategies. The combination of real-time monitoring and historical analysis enables informed decision-making in the fast-paced MEV environment.

Next Steps:
- Set up your own analytics infrastructure with API keys and monitoring systems
- Create custom Dune queries for your specific MEV strategies
- Implement automated opportunity detection for your target MEV types
- Build comprehensive dashboards for tracking your MEV performance

Remember: Effective analytics require continuous monitoring, data validation, and strategy refinement. The blockchain environment evolves rapidly, so your analytics tools must adapt accordingly.