Matching Algorithm Specification - REG NMS Inspired Design

---

1. Executive Summary

This document details the core matching algorithm implementation of the cryptocurrency matching engine, designed with REG NMS-inspired principles for price-time priority execution and internal order protection. The system implements a comprehensive order matching framework that ensures fair, efficient, and transparent trade execution while maintaining strict price-time priority rules.

1.1 Design Philosophy

The matching algorithm prioritizes market integrity through:

- **Price-Time Priority**: Orders at better prices execute first; at equal prices, earlier orders execute first
- **Internal Order Protection**: Incoming marketable orders must execute at the best available internal price
- **Trade-Through Prevention**: System prevents execution at prices inferior to the best bid/offer
- **Order Type Compliance**: Each order type (Market, Limit, IOC, FOK) follows specific execution semantics

1.2 Core Components

Primary Implementation Files:

- `engine/book.py`: Core matching logic in `_match_order()` method, price-time priority enforcement
- `engine/order.py`: Order type definitions and behavior specifications
- `engine/matcher.py`: Order validation, trade execution orchestration, and event management

---

2. BBO Calculation and Dissemination

2.1 Best Bid/Offer Maintenance

Implementation Location: `engine/book.py:185-205`

The system maintains real-time Best Bid and Offer (BBO) calculations using efficient data structures optimized for frequent updates and queries.

2.1.1 Data Structure Design

```python
# Bid storage: SortedDict with negative keys for descending price order
self.bids: SortedDict[Decimal, PriceLevel] = SortedDict()

# Ask storage: SortedDict with positive keys for ascending price order
self.asks: SortedDict[Decimal, PriceLevel] = SortedDict()
```

Rationale: SortedDict provides O(log n) insertion/deletion with O(1) access to best prices, enabling real-time BBO updates without full book traversal.

2.1.2 BBO Calculation Algorithm

```python
def best_bid(self) -> Optional[Decimal]:
    """Get best bid price (highest)."""
    with self._lock:
        return -self.bids.peekitem(-1)[0] if self.bids else None

def best_ask(self) -> Optional[Decimal]:
    """Get best ask price (lowest)."""
    with self._lock:
        return self.asks.peekitem(0)[0] if self.asks else None
```

Key Features:

- **Thread Safety**: All BBO operations protected by RLock for concurrent access
- **Constant Time Access**: Best prices retrieved in O(1) using SortedDict peek operations
- **Null Safety**: Graceful handling of empty book conditions

2.1.3 Real-Time BBO Updates

The BBO is recalculated and disseminated on every order book state change:

Triggering Events:

1. **Order Addition**: New resting orders may improve BBO
2. **Order Cancellation**: Removing best price orders triggers BBO recalculation
3. **Trade Execution**: Fills may consume best price levels
4. **Order Modification**: Price/quantity changes affect BBO

Update Flow:

```
Order Event → Book State Change → BBO Recalculation → Event Emission → Client Broadcast
```

2.2 Spread and Mid-Price Calculations

Implementation: `engine/book.py:193-205`

```python
def spread(self) -> Optional[Decimal]:
    """Calculate bid-ask spread."""
    bb, ba = self.best_bid(), self.best_ask()
    return ba - bb if bb and ba else None

def mid_price(self) -> Optional[Decimal]:
    """Calculate mid price."""
    bb, ba = self.best_bid(), self.best_ask()
    return (bb + ba) / 2 if bb and ba else None
```

Market Data Output Format:

```json
{
  "symbol": "BTC-USDT",
  "best_bid": "30095.00",
  "best_ask": "30100.00",
  "spread": "5.00",
  "mid_price": "30097.50",
  "timestamp": 1640995200.123456
}
```

---

3. Price-Time Priority Algorithm

3.1 Price Level Management

Implementation: `engine/book.py:PriceLevel` class (lines 15-87)

Each price level maintains orders in strict time priority using a double-ended queue (deque) with O(1) append/pop operations at both ends.

```python
```

```python
class PriceLevel:
    def __init__(self, price: Decimal):
        self.price: Decimal = price
        self.orders: deque[Order] = deque()  # Time-ordered queue
        self.aggregate: Decimal = Decimal("0")  # Total quantity at price
        self.order_map: Dict[str, Order] = {}  # O(1) order lookup
```

3.1.1 Time Priority Enforcement

Order Addition: New orders appended to end of deque, preserving time sequence

```python
def add_order(self, order: Order) -> None:
    with self._lock:
        self.orders.append(order)  # Maintains time priority
        self.order_map[order.order_id] = order
        self.aggregate += order.remaining
```

Order Matching: Always processes oldest order first via `pop_oldest()`

```python
def pop_oldest(self) -> Optional[Order]:
    with self._lock:
        if not self.orders:
            return None
        order = self.orders.popleft()  # FIFO processing
        self.order_map.pop(order.order_id, None)
        self.aggregate -= order.remaining
        return order
```

3.2 Cross-Market Matching Logic

Implementation: `engine/book.py:_match_order()` method (lines 415-520)

The core matching algorithm processes incoming orders against the opposite side of the book, ensuring strict price-time priority compliance.

3.2.1 Matching Sequence

```python
```

```python
def _match_order(self, incoming_order: Order) -> List[Trade]:
    trades: List[Trade] = []

    # Determine opposite book side
    if incoming_order.side == "buy":
        opposite_levels = self.asks  # Buyers take asks
        price_check = lambda p: incoming_order.is_market() or incoming_order.price >= p
    else:
        opposite_levels = self.bids  # Sellers take bids
        price_check = lambda p: incoming_order.is_market() or incoming_order.price <= (-p)
```

3.2.2 Price Improvement Logic

Price Traversal Order:

- **Buy Orders**: Process asks from lowest to highest price (best price first)
- **Sell Orders**: Process bids from highest to lowest price (best price first)

Trade Execution Price: All trades execute at the resting order's price, providing price improvement to the aggressor when possible.

```python
# Execute trade at the resting order's price (price improvement for aggressor)
trade = self._execute_trade(maker_order, incoming_order, trade_qty, level_price)
```

3.3 Internal Order Protection

The system implements comprehensive trade-through prevention to ensure incoming marketable orders receive execution at the best available internal prices.

3.3.1 Marketability Determination

```python
def _is_order_marketable(self, order: Order) -> bool:
    """Check if order can be immediately matched."""
    if order.is_market():
        return True

    if order.side == "buy":
        best_ask = self.best_ask()
        return best_ask is not None and order.price >= best_ask
    else:
        best_bid = self.best_bid()
        return best_bid is not None and order.price <= best_bid
```

3.3.2 Trade-Through Prevention

Validation Process:

1. **Price Crossing Check**: Verify incoming limit order can execute at or better than specified price
2. **Liquidity Validation**: For FOK orders, pre-validate sufficient aggregate liquidity exists
3. **Sequential Matching**: Process price levels in strict price priority order
4. **Internal Execution**: All matching occurs within internal book before considering external venues

---

4. Order Type Handling

4.1 Market Order Processing

Execution Characteristics:

- **Immediate Execution**: Matches against best available prices immediately
- **Price Agnostic**: Accepts current market prices without limit
- **Partial Fill Handling**: Cancels unfilled quantity if insufficient liquidity

Implementation Logic:

```python
if order.order_type == "market":
    # Market order remainder cancelled (no more liquidity)
    order.cancel()
    logger.warning(f"Market order {order.order_id} partially cancelled: no liquidity")
```

4.2 Limit Order Processing

Execution Characteristics:

- **Price Protection**: Only executes at specified price or better
- **Resting Capability**: Unfilled portions remain on order book
- **Price-Time Priority**: Maintains queue position based on submission time

Implementation Flow:

```python
```

```python
elif order.order_type == "limit":
    # Rest limit order on book
    self._add_resting_order(order)
    order_resting = True
```

4.3 Immediate-or-Cancel (IOC) Processing

Execution Characteristics:

- **Immediate Execution Only**: Matches available quantity immediately
- **No Resting**: Cancels any unfilled quantity immediately
- **Price Protection**: Respects specified limit price

Implementation Logic:

```python
elif order.order_type == "ioc":
    # IOC remainder cancelled
    order.cancel()
```

4.4 Fill-or-Kill (FOK) Processing

Execution Characteristics:

- **All-or-Nothing**: Either fills completely or cancels entirely
- **Liquidity Pre-Validation**: Checks sufficient liquidity before execution
- **Atomic Execution**: No partial fills allowed

Implementation with Liquidity Check:

```python
if incoming_order.is_fok():
    max_price = incoming_order.price if not incoming_order.is_market() else None
    available = self._calculate_available_liquidity(incoming_order.side, max_price)
    if available < incoming_order.remaining:
        logger.info(f"FOK order {incoming_order.order_id} rejected: insufficient liquidity")
        incoming_order.reject("Insufficient liquidity")
        return []
```

---

5. Trade Execution Engine

5.1 Trade Generation Process

Implementation: `engine/book.py:_execute_trade()` method (lines 370-400)

Each successful order match generates a Trade object containing complete execution details for audit trail and reporting purposes.

```python
def _execute_trade(self, maker_order: Order, taker_order: Order,
                   quantity: Decimal, price: Decimal) -> Trade:
    # Update order quantities
    maker_order.fill(quantity)
    taker_order.fill(quantity)

    # Update price level aggregate
    if maker_order.order_id in self.orders:
        _, level = self.orders[maker_order.order_id]
        level.update_order_fill(maker_order.order_id, quantity)

    # Create trade record
    self.trade_seq += 1
    trade = Trade(
        symbol=self.symbol,
        price=price,
        quantity=quantity,
        maker_order=maker_order,
        taker_order=taker_order,
        trade_seq=self.trade_seq
    )
```

5.2 Trade Sequencing and Identification

Unique Trade ID Generation:

```python
self.trade_id = f"{symbol}-{trade_seq}-{str(uuid.uuid4())[:8]}"
```

Components:

- **Symbol**: Trading pair identifier for market segmentation
- **Sequence Number**: Monotonic sequence per symbol ensuring chronological ordering
- **UUID Fragment**: Additional entropy for global uniqueness across restarts

5.3 Audit Trail Generation

Trade Record Structure:

```python
def to_dict(self) -> dict:
    return {
        "timestamp": self.timestamp,
        "symbol": self.symbol,
        "trade_id": self.trade_id,
        "price": str(self.price),
        "quantity": str(self.quantity),
        "aggressor_side": self.aggressor_side,
        "maker_order_id": self.maker_order_id,
        "taker_order_id": self.taker_order_id
    }
```

Audit Features:

- **Microsecond Timestamps**: High-precision execution timing
- **Order Correlation**: Links trades to originating orders for complete transaction history
- **Aggressor Identification**: Distinguishes market takers from liquidity providers
- **Price-Quantity Precision**: Decimal precision maintained throughout execution chain

---

6. Concurrency and Thread Safety

6.1 Locking Strategy

The current implementation uses a simplified locking approach prioritizing correctness over maximum performance:

Engine Level: Single RLock in `matcher.py` for engine-wide operations

```python
with self._lock:
    # Get book and submit order
    book = self._get_book(symbol)
    trades, order_resting = book.submit_order(order)
```

Book Level: Individual RLock per OrderBook for book-specific operations

```python
with self._lock:
    # Validate order and attempt matching
    trades = self._match_order(incoming_order)
```

Price Level: RLock per PriceLevel for granular order management

```python
with self._lock:
    self.orders.append(order)
    self.aggregate += order.remaining
```

6.2 Deadlock Prevention

Lock Ordering: Consistent lock acquisition order prevents circular dependencies:

1. Engine lock (if needed)
2. Book lock
3. Price level locks (acquired as needed during matching)

Lock Scope Minimization: Critical sections kept as small as possible to reduce contention:

```python
# Good: Minimal lock scope
with self._lock:
    trades = self._match_order(order)
# Event processing outside lock
for trade in trades:
    self._emit_trade_event(trade)
```

6.3 Performance Considerations

Current Limitations:

- Global engine lock creates bottleneck for multi-symbol operations
- Nested locking can cause performance degradation under high load
- Synchronous event emission blocks order processing

Optimization Opportunities:

- Per-symbol locking to enable parallel processing
- Lock-free data structures for read-heavy operations
- Asynchronous event emission to decouple matching from notifications

---

7. Algorithm Performance Analysis

7.1 Computational Complexity

Order Submission: O(log n + k)

- O(log n): SortedDict price level lookup/insertion
- O(k): Matching against k orders at price levels

Order Cancellation: O(log n + m)

- O(log n): Price level lookup
- O(m): Linear search within price level deque (unavoidable with current structure)

BBO Calculation: O(1)

- Constant time access to best prices via SortedDict peek operations

Book Snapshot: O(n)

- Linear traversal of top n price levels

7.2 Memory Utilization

Order Storage: Each order maintains references in:

- PriceLevel.orders deque
- PriceLevel.order_map dictionary
- OrderBook.orders tracking dictionary
- OrderBook.client_order_map (if client ID provided)

Space Complexity: O(n) where n = total active orders across all price levels

7.3 Scalability Characteristics

Throughput Bottlenecks:

- Single-threaded order processing due to global locking
- Event emission synchronously blocks matching pipeline
- Memory allocation for Trade objects during high-frequency matching

Current Performance: 375 orders/sec observed in load testing (target: 1000+ orders/sec)

---

8. Algorithm Validation and Testing

8.1 REG NMS Compliance Verification

Price-Time Priority Testing:

- Orders at same price level execute in submission time order
- Better-priced orders always execute before worse-priced orders
- No trade-through violations occur during matching

Internal Order Protection Testing:

- Marketable orders always receive best available internal price
- FOK orders correctly validate liquidity across multiple price levels
- Order types behave according to specifications

8.2 Edge Case Handling

Empty Book Scenarios:

- Market orders in empty books are cancelled with appropriate logging
- BBO calculations return null for empty books without errors

Precision Handling:

- Decimal arithmetic prevents floating-point rounding errors
- Tick size validation ensures price conformance
- Quantity validation enforces minimum size requirements

8.3 Error Recovery

Order Rejection Handling:

- Invalid orders rejected with detailed error messages
- Order state consistently maintained during error conditions
- Transaction atomicity preserved during matching failures

---

9. Future Enhancements

9.1 Performance Optimizations

Lock-Free Structures: Replace synchronized collections with lock-free alternatives for read-heavy operations

Per-Symbol Threading: Eliminate global locks to enable parallel processing of different trading pairs

Batch Processing: Group multiple orders for atomic processing to reduce locking overhead

9.2 Algorithm Extensions

Hidden Order Support: Implement iceberg and reserve orders with partial quantity disclosure

Advanced Order Types: Add stop-loss, stop-limit, and trailing stop functionality

Cross-Symbol Arbitrage: Implement multi-leg order support for complex trading strategies

9.3 Monitoring and Analytics

Latency Profiling: Add microsecond-precision timing for each algorithm stage

Fairness Metrics: Implement queue-time analytics to verify time priority enforcement

Market Quality Indicators: Calculate spread stability, market depth, and execution quality metrics

---

Document End

This specification provides complete algorithmic detail for the REG NMS-inspired matching engine, ensuring compliance with regulatory principles while maintaining high-performance execution characteristics.