# PeeGeeQ Outbox: Partitioned Ordering - Complete Guide

**Author**: Mark A Ray-Smith, Cityline Ltd  
**Date**: October 2025  
**Status**: Production Ready  
**Version**: 1.0

## Executive Summary

**Key Finding**: PeeGeeQ outbox **DOES support partitioned ordering** through the message\_group column, but requires explicit application-level configuration to guarantee strict ordering.

### Current Capabilities

| Feature | Status | Details |
| --- | --- | --- |
| Message Group Column | ✅ | Stored in outbox table, indexed |
| Producer API | ✅ | All send methods accept messageGroup parameter |
| Consumer Retrieval | ✅ | Message group returned with each message |
| FIFO Ordering | ✅ | Messages ordered by created\_at ASC |
| Per-Group Storage | ✅ | Messages with same group stored in order |
| Automatic Partition Assignment | ❌ | Must implement in application |
| Partition-Aware Consumer Distribution | ❌ | No built-in partition routing |
| Automatic Rebalancing | ❌ | No failover/rebalancing logic |

## Ordering Guarantees

### Global FIFO Ordering

**Guarantee**: All messages processed in created\_at ASC order.

**Applies To**: Entire topic, all message groups.

**Limitation**: With multiple concurrent consumers, processing order may differ from storage order.

### Per-Group Ordering

**Guarantee**: Messages with same message\_group are **stored** in order.

**NOT Guaranteed**: Messages with same message\_group are **processed** in order when multiple consumers process the same group concurrently.

**Solution**: Use single consumer per group for strict ordering.

## Three Patterns for Strict Ordering

### Pattern 1: Single Consumer Per Partition ⭐ Recommended

**Use When**: < 100 partitions, throughput < 10,000 msg/s

// Producer: Set partition key  
producer.send(payload, headers, correlationId, "customer-123");  
  
// Consumer: Single consumer processes all messages  
MessageConsumer<OrderEvent> consumer = factory.createConsumer(topic, OrderEvent.class);  
consumer.subscribe(message -> {  
 OrderEvent event = message.getPayload();  
 String partition = message.getMessageGroup();  
 logger.info("Processing order for customer {}", partition);  
 return processOrder(event);  
});  
consumer.start();

**Guarantees**: ✅ Strict ordering per partition  
**Throughput**: ~1,000 msg/s per consumer  
**Complexity**: Low  
**Scalability**: O(partitions)

### Pattern 2: Consumer Group with Partition Filter

**Use When**: 10-100 partitions, throughput 1,000-10,000 msg/s

// Producer: Set partition key  
producer.send(payload, headers, correlationId, "security-456");  
  
// Consumer: Filter by partition  
ConsumerGroup<TradeEvent> group = factory.createConsumerGroup(  
 "settlement", topic, TradeEvent.class);  
  
group.addConsumer("settlement-1", message -> {  
 TradeEvent trade = message.getPayload();  
 return processSettlement(trade);  
}, message -> message.getMessageGroup().startsWith("security"));  
  
group.start();

**Guarantees**: ✅ Strict ordering per partition  
**Throughput**: ~5,000 msg/s total  
**Complexity**: Medium  
**Scalability**: O(partitions)

### Pattern 3: Application-Level Partitioning

**Use When**: > 100 partitions, dynamic partitions, complex logic

public class PartitionAwareConsumer {  
 private final Map<String, LinkedBlockingQueue<Message<?>>> partitionQueues   
 = new ConcurrentHashMap<>();  
 private final ExecutorService partitionExecutors = Executors.newCachedThreadPool();  
   
 public void startPartitionedConsumer(MessageConsumer<OrderEvent> consumer) {  
 consumer.subscribe(message -> {  
 String partition = message.getMessageGroup();  
   
 // Route to partition-specific queue  
 partitionQueues.computeIfAbsent(partition, k ->   
 new LinkedBlockingQueue<>()).offer(message);  
   
 // Process partition queue (single thread per partition)  
 processPartitionQueue(partition);  
   
 return CompletableFuture.completedFuture(null);  
 });  
 }  
   
 private void processPartitionQueue(String partition) {  
 partitionExecutors.execute(() -> {  
 LinkedBlockingQueue<Message<?>> queue = partitionQueues.get(partition);  
 while (!Thread.currentThread().isInterrupted()) {  
 try {  
 Message<?> message = queue.poll(1, TimeUnit.SECONDS);  
 if (message != null) {  
 processMessage(message);  
 }  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 }  
 }  
 });  
 }  
}

**Guarantees**: ✅ Strict ordering per partition  
**Throughput**: ~5,000 msg/s total  
**Complexity**: High  
**Scalability**: O(partitions)

## Database Schema

### Outbox Table Structure

CREATE TABLE outbox (  
 id BIGSERIAL PRIMARY KEY,  
 topic VARCHAR(255) NOT NULL,  
 payload JSONB NOT NULL,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),  
 processed\_at TIMESTAMP WITH TIME ZONE,  
 status VARCHAR(50) DEFAULT 'PENDING',  
 headers JSONB DEFAULT '{}',  
 correlation\_id VARCHAR(255),  
 message\_group VARCHAR(255), -- Partition key  
 priority INT DEFAULT 5,  
 retry\_count INT DEFAULT 0,  
 max\_retries INT DEFAULT 3  
);  
  
-- Indexes for ordering  
CREATE INDEX idx\_outbox\_topic\_status ON outbox(topic, status);  
CREATE INDEX idx\_outbox\_status\_created\_at ON outbox(status, created\_at);  
CREATE INDEX idx\_outbox\_message\_group ON outbox(message\_group)   
 WHERE message\_group IS NOT NULL;

### Current Consumer Query

String sql = """  
 UPDATE outbox  
 SET status = 'PROCESSING', processed\_at = $1  
 WHERE id IN (  
 SELECT id FROM outbox  
 WHERE topic = $2 AND status = 'PENDING'  
 ORDER BY created\_at ASC  
 LIMIT $3  
 FOR UPDATE SKIP LOCKED  
 )  
 RETURNING id, payload, headers, correlation\_id, message\_group, created\_at  
 """;

**Key Points**: - Orders by created\_at ASC (FIFO) - Does NOT filter by message\_group - Uses FOR UPDATE SKIP LOCKED for concurrent safety - Returns message\_group for consumer use

## Use Case Scenarios

### Scenario 1: E-Commerce Order Processing

**Requirements**: - Orders from same customer must be processed in order - Different customers can be processed in parallel - High throughput (10,000+ orders/day)

**Solution**: Single Consumer Per Partition

producer.send(order, headers, correlationId, order.getCustomerId());

**Guarantees**: ✅ Order 1 → Order 2 → Order 3 (per customer)

### Scenario 2: Financial Trade Settlement

**Requirements**: - Trades for same security must be settled in order - Different securities can be settled in parallel - Strict ordering critical for regulatory compliance

**Solution**: Consumer Group with Partition Filter

producer.send(trade, headers, correlationId, trade.getSecurityId());

**Guarantees**: ✅ Strict ordering per security

### Scenario 3: Real-Time Analytics

**Requirements**: - Process all events as fast as possible - Ordering not critical - Maximize throughput

**Solution**: Multiple Consumers (No Partitioning)

producer.send(event, headers, correlationId, null);

**Guarantees**: ✅ Maximum throughput (10x single consumer)

### Scenario 4: Saga Orchestration

**Requirements**: - Saga steps must execute in order - Multiple sagas can run in parallel - Compensation on failure

**Solution**: Application-Level Partitioning

producer.send(sagaStep, headers, correlationId, saga.getId());

**Guarantees**: ✅ Saga steps in order

## Decision Matrix

### Do You Need Strict Ordering?

Question 1: Must messages be processed in exact order?  
├─ YES → Question 2  
└─ NO → Use Multiple Consumers (High Throughput)  
  
Question 2: Is ordering required per customer/entity or globally?  
├─ Per Entity → Question 3  
├─ Globally → Use Single Consumer (Low Throughput)  
└─ Not Sure → Consult architect  
  
Question 3: How many entities (partitions) do you have?  
├─ < 100 → Pattern 1: Single Consumer Per Partition  
├─ 10-100 → Pattern 2: Consumer Group with Partition Filter  
├─ > 100 → Pattern 3: Application-Level Partitioning  
└─ Dynamic → Pattern 3: Application-Level Partitioning

## Performance Characteristics

| Pattern | Throughput | Latency | Ordering | Complexity |
| --- | --- | --- | --- | --- |
| Multiple Consumers | 10,000+ msg/s | <5ms | None | Low |
| Single Consumer/Partition | 1,000 msg/s | <10ms | Strict | Low |
| Consumer Group + Filter | 5,000 msg/s | <10ms | Strict | Medium |
| Application-Level | 5,000 msg/s | 10-50ms | Strict | High |

## Best Practices

### ✅ DO

1. **Define Partition Key**: Choose entity that requires ordering
   * Customer ID, Security ID, Account ID, Saga ID
2. **Document Strategy**: Record partition key and rationale
   * Why this key? How many partitions? Expected throughput?
3. **Set Message Group**: Always populate message\_group in producer
   * producer.send(payload, headers, correlationId, partitionKey)
4. **Monitor Ordering**: Verify messages processed in order
   * Track sequence numbers per partition
   * Alert on out-of-order messages
5. **Test Thoroughly**: Load test with concurrent sends
   * Verify ordering under load
   * Test consumer failures and recovery

### ❌ DON’T

1. **Assume Ordering with Multiple Consumers**: Requires single consumer per partition
2. **Ignore Message Group**: Set it even if not using partitioning
3. **Mix Ordered and Unordered**: Choose pattern and stick with it
4. **Forget Monitoring**: Ordering issues discovered in production are expensive
5. **Ignore Partition Imbalance**: Uneven distribution causes bottlenecks

## Implementation Checklist

### Pre-Implementation

* Identify ordering requirement (YES/NO)
* Determine if ordering per entity or global
* Identify partition key (customer, security, account, etc.)
* Estimate partition count
* Define throughput requirement
* Choose pattern (1, 2, or 3)
* Document architecture

### Implementation

* Update producer to set message\_group
* Implement consumer for chosen pattern
* Add partition key extraction logic
* Implement message handler
* Add error handling
* Add logging and monitoring

### Testing

* Unit tests for partition key extraction
* Integration tests for ordering
* Load tests with concurrent sends
* Failure tests (consumer, producer, database)
* Performance tests (throughput, latency)

### Deployment

* Code review and approval
* Staging deployment
* Production deployment
* Monitor closely
* Verify ordering
* Check metrics and alerts

### Post-Deployment

* Daily monitoring
* Weekly performance review
* Monthly optimization
* Document lessons learned

## Monitoring and Observability

### Key Metrics Per Partition

1. Message Count  
 - Total messages per partition  
 - Pending messages per partition  
 - Failed messages per partition  
  
2. Latency  
 - Time in queue per partition  
 - Processing time per partition  
 - End-to-end latency per partition  
  
3. Throughput  
 - Messages/sec per partition  
 - Bytes/sec per partition  
 - Success rate per partition  
  
4. Ordering  
 - Out-of-order messages (should be 0)  
 - Sequence gaps (should be 0)  
 - Reordering events (should be 0)

### Alerting Rules

* Alert if out-of-order messages > 0
* Alert if partition queue depth > threshold
* Alert if partition latency > SLA
* Alert if partition processing fails
* Alert if partition consumer down

## Common Mistakes to Avoid

### ❌ Mistake 1: Assuming Ordering with Multiple Consumers

// WRONG: Multiple consumers, no partition awareness  
ConsumerGroup<Order> group = factory.createConsumerGroup(  
 "orders", topic, Order.class);  
group.addConsumer("c1", this::processOrder);  
group.addConsumer("c2", this::processOrder);  
// Result: Orders from same customer may be out of order!

### ❌ Mistake 2: Not Setting Message Group

// WRONG: No partition key  
producer.send(order, headers, correlationId, null);  
// Result: No way to enforce ordering

### ❌ Mistake 3: Ignoring Partition Imbalance

1000 orders for customer-1  
1 order for customer-2  
Result: Customer-1 consumer overloaded

### ❌ Mistake 4: No Monitoring

No visibility into ordering  
Result: Discover ordering issues in production

## Troubleshooting

### Problem: Messages Out of Order

**Diagnosis**: 1. Check if multiple consumers processing same partition 2. Verify message\_group is set correctly 3. Check consumer logs for concurrent processing 4. Monitor partition queue depth

**Solution**: - Use single consumer per partition - Implement application-level ordering - Add ordering verification tests

### Problem: High Latency

**Diagnosis**: 1. Check partition distribution 2. Monitor consumer processing time 3. Check database query performance 4. Verify network latency

**Solution**: - Add more consumers (if no ordering required) - Optimize processing logic - Increase batch size - Add database indexes

## Future Enhancements

### Phase 2 (Recommended)

* Partition-aware consumer query optimization
* Consumer group partition assignment API
* Per-partition metrics and monitoring
* Partition rebalancing support

### Phase 3 (Future)

* Ordered consumer group variant
* Automatic partition assignment
* Partition rebalancing protocol
* Kafka-compatible partition API

## Summary

| Aspect | Current | Recommended |
| --- | --- | --- |
| **Message Group Storage** | ✅ Supported | Use for logical partitioning |
| **Per-Group Ordering** | ⚠️ Partial | Single consumer per group |
| **Automatic Partitioning** | ❌ Not supported | Implement in application |
| **Rebalancing** | ❌ Not supported | Use external service discovery |
| **Strict Ordering** | ✅ Possible | With single consumer per group |

**Conclusion**: PeeGeeQ outbox supports partitioned ordering through message groups, but requires explicit application-level configuration to guarantee strict ordering. For production systems requiring strict per-partition ordering, implement one of the three patterns based on your partition count and throughput requirements.

## Ordering Verification Test

@Test  
void testPartitionedOrdering() {  
 Map<String, List<Integer>> partitionSequences = new ConcurrentHashMap<>();  
  
 consumer.subscribe(message -> {  
 OrderEvent event = message.getPayload();  
 String partition = message.getMessageGroup();  
 int sequence = event.getSequence();  
  
 // Track sequence per partition  
 partitionSequences.computeIfAbsent(partition, k ->  
 Collections.synchronizedList(new ArrayList<>()))  
 .add(sequence);  
  
 return CompletableFuture.completedFuture(null);  
 });  
  
 // Send messages with different partitions  
 for (int i = 1; i <= 100; i++) {  
 String partition = "customer-" + (i % 10);  
 OrderEvent event = new OrderEvent("order-" + i, i, partition);  
 producer.send(event, null, null, partition).join();  
 }  
  
 // Verify ordering per partition  
 for (List<Integer> sequences : partitionSequences.values()) {  
 for (int i = 1; i < sequences.size(); i++) {  
 assertTrue(sequences.get(i) > sequences.get(i-1),  
 "Sequence out of order: " + sequences);  
 }  
 }  
}

## Producer Implementation Details

### Send with Message Group

public CompletableFuture<Void> send(T payload, Map<String, String> headers,  
 String correlationId, String messageGroup) {  
 return sendInternal(payload, headers, correlationId, messageGroup);  
}  
  
// Internal implementation  
String sql = """  
 INSERT INTO outbox (topic, payload, headers, correlation\_id, message\_group,  
 created\_at, status)  
 VALUES ($1, $2::jsonb, $3::jsonb, $4, $5, $6, 'PENDING')  
 """;  
  
Tuple params = Tuple.of(  
 topic,  
 payloadJson,  
 headersJson,  
 finalCorrelationId,  
 messageGroup, // Partition key  
 OffsetDateTime.now()  
);

### Transactional Send with Message Group

public CompletableFuture<Void> sendWithTransaction(T payload,  
 Map<String, String> headers,  
 String correlationId,  
 String messageGroup,  
 SqlConnection connection) {  
 String sql = """  
 INSERT INTO outbox (topic, payload, headers, correlation\_id, message\_group,  
 created\_at, status)  
 VALUES ($1, $2::jsonb, $3::jsonb, $4, $5, $6, 'PENDING')  
 """;  
  
 Tuple params = Tuple.of(  
 topic,  
 toJsonObject(payload),  
 headersToJsonObject(headers),  
 correlationId != null ? correlationId : UUID.randomUUID().toString(),  
 messageGroup,  
 OffsetDateTime.now()  
 );  
  
 return connection.preparedQuery(sql)  
 .execute(params)  
 .mapEmpty()  
 .toCompletionStage()  
 .toCompletableFuture();  
}

## Consumer Implementation Details

### Single Consumer Pattern

public void startSingleConsumer() {  
 MessageConsumer<OrderEvent> consumer = factory.createConsumer(topic, OrderEvent.class);  
  
 consumer.subscribe(message -> {  
 OrderEvent event = message.getPayload();  
 String partition = message.getMessageGroup();  
  
 logger.info("Processing order {} for partition {}",  
 event.getOrderId(), partition);  
  
 try {  
 // Process in strict order - only one consumer  
 return processOrder(event)  
 .thenApply(result -> {  
 logger.info("Successfully processed order {} for partition {}",  
 event.getOrderId(), partition);  
 return null;  
 })  
 .exceptionally(ex -> {  
 logger.error("Failed to process order {} for partition {}: {}",  
 event.getOrderId(), partition, ex.getMessage());  
 throw new RuntimeException(ex);  
 });  
 } catch (Exception e) {  
 logger.error("Error processing order: {}", e.getMessage());  
 return CompletableFuture.failedFuture(e);  
 }  
 });  
  
 consumer.start();  
}

### Consumer Group with Partition Filter

public void startConsumerGroupWithFilter() {  
 ConsumerGroup<OrderEvent> group = factory.createConsumerGroup(  
 "order-processors", topic, OrderEvent.class);  
  
 // Add consumers with partition-specific filters  
 String[] partitions = {"customer-1", "customer-2", "customer-3"};  
  
 for (int i = 0; i < partitions.length; i++) {  
 final String partition = partitions[i];  
 final int consumerId = i;  
  
 group.addConsumer("consumer-" + consumerId,  
 message -> {  
 OrderEvent event = message.getPayload();  
 logger.info("Consumer {} processing order for partition {}",  
 consumerId, partition);  
 return processOrder(event);  
 },  
 message -> partition.equals(message.getMessageGroup())  
 );  
 }  
  
 group.start();  
}

## Requirements Checklist

### Before Implementation

* Identify partition key (customer, security, account, saga, etc.)
* Estimate partition count (< 100, 10-100, > 100)
* Define throughput requirement (msg/s)
* Define latency SLA (ms)
* Choose pattern (1, 2, or 3)
* Document architecture and rationale
* Identify monitoring requirements
* Plan testing strategy

### During Implementation

* Producer sets message\_group correctly
* Consumer retrieves message\_group
* Partition key extraction logic tested
* Error handling implemented
* Logging and monitoring added
* Code reviewed and approved

### After Implementation

* Unit tests passing
* Integration tests passing
* Load tests completed
* Failure tests completed
* Performance meets SLA
* Monitoring working
* Documentation updated
* Team trained