System Design Solutions: Problems 26-45

Complete architectural solutions with diagrams, code examples, and implementation details.

---

26. Distributed Key-Value Store

Problem Overview

Design a distributed key-value data store like Redis or DynamoDB that provides high availability, horizontal scalability, and tunable consistency guarantees.

Back-of-the-Envelope Estimation

- **Operations/sec**: 1 million ops/sec (70% reads, 30% writes)
- **Data size**: 1TB total data
- **Key size**: Average 50 bytes
- **Value size**: Average 1KB
- **Total keys**: ~1 billion keys
- **Nodes**: 10 nodes initially, scalable to 100+
- **Replication factor**: 3 (for durability)
- **Storage per node**: 100GB with replication

Functional Requirements

- **FR1**: Put(key, value) - Store key-value pair
- **FR2**: Get(key) - Retrieve value by key
- **FR3**: Delete(key) - Remove key-value pair
- **FR4**: Support TTL (Time To Live) for automatic expiration
- **FR5**: Atomic operations (increment, compare-and-swap)
- **FR6**: Range queries (scan keys by prefix)

Non-Functional Requirements

- **Scalability**: Horizontal scaling to 100+ nodes
- **Availability**: 99.99% uptime (4 nines)
- **Latency**: <1ms for reads, <5ms for writes (p99)
- **Consistency**: Tunable (strong, eventual, or read-your-writes)
- **Durability**: No data loss (replicated to N nodes)
- **Partition Tolerance**: Continue operating during network partitions

High-Level Architecture

Components:

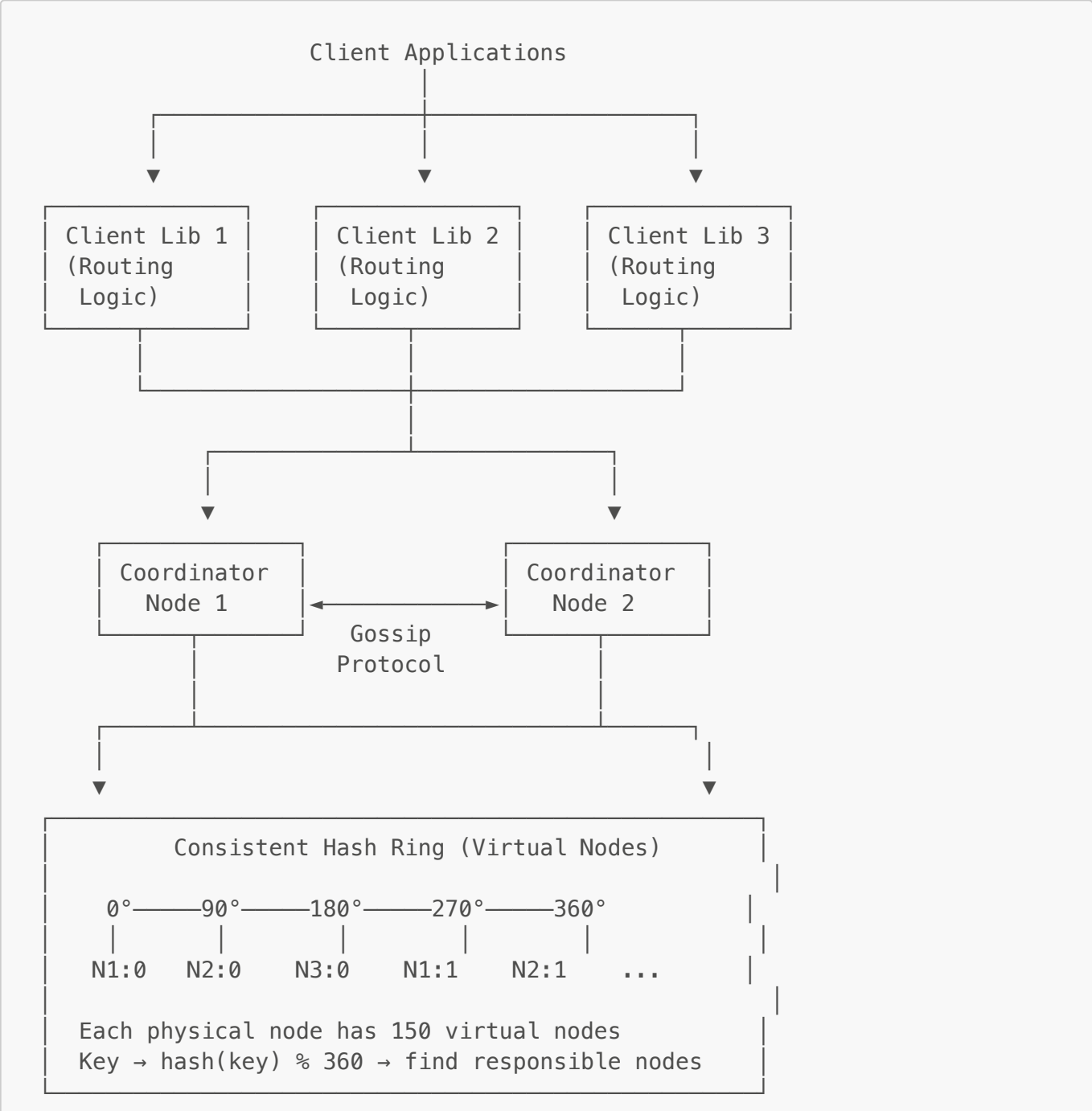
- **Client Library**: Smart client with routing logic
- **Coordinator Nodes**: Handle requests, route to correct partition
- **Storage Nodes**: Store actual data with LSM trees

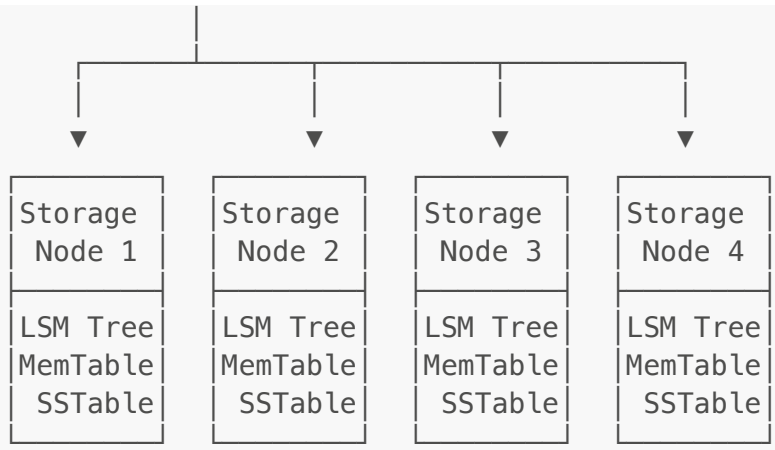
- **Replication Manager**: Ensure N replicas per key
- **Membership Service**: Track node health (gossip protocol)
- **Partition Manager**: Consistent hashing for data distribution

Data Storage Choices

Component	Technology	Justification
Storage Engine	LSM Tree (RocksDB)	High write throughput, efficient compaction
Replication	Multi-master	High availability, low latency writes
Partitioning	Consistent Hashing	Even distribution, minimal data movement
Membership	Gossip Protocol	Decentralized, fault-tolerant

Architecture Diagram:






```
Replication Strategy (N=3):
┌─────────────────────────────────────────┐
│ Key "user:123" → hash → position 45°    │
│                                         │
│ Primary replicas (clockwise):           │
│ 1. Node 2 (45°)  ← Primary/Leader       │
│ 2. Node 3 (90°)  ← Replica              │
│ 3. Node 1 (135°) ← Replica              │
└─────────────────────────────────────────┘

Write Path (Quorum W=2):
1. Client → hash(key) → find replicas [N2, N3, N1]
2. Send write to all 3 replicas in parallel
3. Wait for W=2 acknowledgments
4. Return success to client
5. Async: 3rd replica eventually catches up

Read Path (Quorum R=2):
1. Client → hash(key) → find replicas [N2, N3, N1]
2. Send read to R=2 replicas
3. If values match → return
4. If values differ → read repair (return latest)
```

Implementation:

```java
import java.security.MessageDigest;
import java.time.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.Collectors;

/**
 * Consistent Hashing Ring for distributed data partitioning.
 * Uses virtual nodes to ensure even distribution across physical nodes.
 */
public class ConsistentHashRing {

    private final int virtualNodesPerNode;
    private final TreeMap<Long, String> ring = new TreeMap<>();
```

```
    private final Set<String> physicalNodes = new HashSet<>();

    public ConsistentHashRing(List<String> nodes, int virtualNodesPerNode)
{
        this.virtualNodesPerNode = virtualNodesPerNode;
        nodes.forEach(this::addNode);
    }

    /**
     * Hash a key to position on the ring (0 to 2^32-1).
     */
    private long hash(String key) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] digest = md.digest(key.getBytes());
            // Use first 4 bytes for hash value
            return ((long)(digest[0] & 0xFF) << 24) |
                   ((long)(digest[1] & 0xFF) << 16) |
                   ((long)(digest[2] & 0xFF) << 8) |
                   ((long)(digest[3] & 0xFF));
        } catch (Exception e) {
            return key.hashCode() & 0xFFFFFFFFL;
        }
    }

    /**
     * Add a physical node with its virtual nodes to the ring.
     */
    public void addNode(String node) {
        physicalNodes.add(node);
        for (int i = 0; i < virtualNodesPerNode; i++) {
            String virtualKey = node + ":" + i;
            long hashValue = hash(virtualKey);
            ring.put(hashValue, node);
        }
    }

    /**
     * Remove a node and all its virtual nodes from the ring.
     */
    public void removeNode(String node) {
        physicalNodes.remove(node);
        ring.entrySet().removeIf(entry -> entry.getValue().equals(node));
    }

    /**
     * Get N responsible nodes for a key (traversing clockwise).
     * Used for replication - returns primary + replica nodes.
     */
    public List<String> getNodesForKey(String key, int count) {
        if (ring.isEmpty()) {
            return Collections.emptyList();
        }
```

```
        long hashValue = hash(key);
        List<String> nodes = new ArrayList<>();
        Set<String> seen = new HashSet<>();

        // Get entries starting from hash position (clockwise)
        SortedMap<Long, String> tailMap = ring.tailMap(hashValue);

        // First check entries after hash position
        for (String node : tailMap.values()) {
            if (!seen.contains(node)) {
                nodes.add(node);
                seen.add(node);
                if (nodes.size() == count) break;
            }
        }

        // Wrap around to beginning if needed
        if (nodes.size() < count) {
            for (String node : ring.values()) {
                if (!seen.contains(node)) {
                    nodes.add(node);
                    seen.add(node);
                    if (nodes.size() == count) break;
                }
            }
        }

        return nodes;
    }
}

/**
 * Distributed Key-Value Store with tunable consistency.
 * Supports quorum reads/writes and automatic replication.
 */
public class DistributedKVStore {

    private final String nodeId;
    private final ConsistentHashRing hashRing;
    private final LSMTreeStorage localStorage;
    private final Map<String, NodeConnection> nodeConnections;

    private final int replicationFactor;  // N
    private final int writeQuorum;        // W
    private final int readQuorum;         // R
    // Consistency guarantee: R + W > N ensures strong consistency

    public DistributedKVStore(String nodeId, List<String> allNodes,
                              int replicationFactor, int writeQuorum, int
readQuorum) {
        this.nodeId = nodeId;
        this.replicationFactor = replicationFactor;
        this.writeQuorum = writeQuorum;
        this.readQuorum = readQuorum;
```

```
        this.hashRing = new ConsistentHashRing(allNodes, 150);
        this.localStorage = new LSMTreeStorage();
        this.nodeConnections = allNodes.stream()
            .collect(Collectors.toMap(n -> n, NodeConnection::new));
    }

    /**
     * Store a key-value pair with optional TTL.
     * Writes to W replicas before returning success.
     */
    public CompletableFuture<Boolean> put(String key, Object value,
Duration ttl) {
        List<String> replicas = hashRing.getNodesForKey(key,
replicationFactor);

        VersionedValue versionedValue = new VersionedValue(
            value,
            Instant.now(),
            generateVersion(),
            ttl
        );

        // Write to all replicas in parallel
        List<CompletableFuture<Boolean>> writeFutures = replicas.stream()
            .map(replica -> {
                if (replica.equals(nodeId)) {
                    return localPut(key, versionedValue);
                } else {
                    return remotePut(replica, key, versionedValue);
                }
            })
            .collect(Collectors.toList());

        // Wait for quorum acknowledgments
        return CompletableFuture.supplyAsync(() -> {
            int successCount = 0;
            for (CompletableFuture<Boolean> future : writeFutures) {
                try {
                    if (future.get(5, TimeUnit.SECONDS)) {
                        successCount++;
                    }
                } catch (Exception e) {
                    // Write to this replica failed
                }
            }
            return successCount >= writeQuorum;
        });
    }

    /**
     * Retrieve a value by key with quorum read.
     * Performs read repair if replicas have inconsistent data.
     */
    public CompletableFuture<Optional<Object>> get(String key) {
```

```
        List<String> replicas = hashRing.getNodesForKey(key,
replicationFactor);

        // Read from R replicas
        List<CompletableFuture<VersionedValue>> readFutures =
            replicas.subList(0, Math.min(readQuorum,
replicas.size())).stream()
                .map(replica -> {
                    if (replica.equals(nodeId)) {
                        return localGet(key);
                    } else {
                        return remoteGet(replica, key);
                    }
                })
                .collect(Collectors.toList());

        return CompletableFuture.supplyAsync(() -> {
            List<VersionedValue> results = new ArrayList<>();

            for (CompletableFuture<VersionedValue> future : readFutures) {
                try {
                    VersionedValue value = future.get(5,
TimeUnit.SECONDS);
                    if (value != null) {
                        results.add(value);
                    }
                } catch (Exception e) {
                    // Read from this replica failed
                }
            }

            if (results.isEmpty()) {
                return Optional.empty();
            }

            // Resolve conflicts - last-write-wins
            VersionedValue latest = results.stream()
                .max(Comparator.comparing(VersionedValue::getTimestamp))
                .orElse(null);

            // Check TTL expiration
            if (latest != null && latest.isExpired()) {
                delete(key);
                return Optional.empty();
            }

            // Read repair if versions differ
            Set<String> versions = results.stream()
                .map(VersionedValue::getVersion)
                .collect(Collectors.toSet());

            if (versions.size() > 1 && latest != null) {
                readRepair(key, latest, replicas);
            }
```

```
            return Optional.ofNullable(latest != null ? latest.getValue()
: null);
        });
    }

    /**
     * Delete a key using tombstone for eventual consistency.
     */
    public CompletableFuture<Boolean> delete(String key) {
        VersionedValue tombstone = new VersionedValue(
            null, Instant.now(), generateVersion(), null, true
        );
        return put(key, tombstone, null);
    }

    private CompletableFuture<Boolean> localPut(String key, VersionedValue
value) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                localStorage.put(key, value);
                return true;
            } catch (Exception e) {
                return false;
            }
        });
    }

    private CompletableFuture<VersionedValue> localGet(String key) {
        return CompletableFuture.supplyAsync(() -> localStorage.get(key));
    }

    private CompletableFuture<Boolean> remotePut(String node, String key,
                                                 VersionedValue value) {
        return nodeConnections.get(node).put(key, value);
    }

    private CompletableFuture<VersionedValue> remoteGet(String node,
String key) {
        return nodeConnections.get(node).get(key);
    }

    private void readRepair(String key, VersionedValue correctValue,
                            List<String> replicas) {
        // Asynchronously repair stale replicas
        for (String replica : replicas) {
            if (!replica.equals(nodeId)) {
                remotePut(replica, key, correctValue);
            }
        }
    }

    private String generateVersion() {
        return nodeId + ":" + Instant.now().toEpochMilli();
```

```
    }
}

/**
 * Versioned value with timestamp, TTL, and tombstone support.
 */
@Data
@AllArgsConstructor
public class VersionedValue {
    private Object value;
    private Instant timestamp;
    private String version;
    private Duration ttl;
    private boolean tombstone;

    public VersionedValue(Object value, Instant timestamp, String version,
Duration ttl) {
        this(value, timestamp, version, ttl, false);
    }

    public boolean isExpired() {
        if (ttl == null) return false;
        return Instant.now().isAfter(timestamp.plus(ttl));
    }
}

/**
 * LSM Tree storage engine (simplified implementation).
 * Provides high write throughput with log-structured storage.
 */
public class LSMTreeStorage {

    private final ConcurrentHashMap<String, VersionedValue> memtable =
        new ConcurrentHashMap<>();
    private final List<SSTable> sstables =
        Collections.synchronizedList(new ArrayList<>());
    private static final int MEMTABLE_SIZE_LIMIT = 1000;

    public void put(String key, VersionedValue value) {
        memtable.put(key, value);

        if (memtable.size() >= MEMTABLE_SIZE_LIMIT) {
            flushMemtable();
        }
    }

    public VersionedValue get(String key) {
        // Check memtable first (most recent data)
        VersionedValue value = memtable.get(key);
        if (value != null) return value;

        // Check SSTables from newest to oldest
        for (int i = sstables.size() - 1; i >= 0; i--) {
            value = sstables.get(i).get(key);
```

```
            if (value != null) return value;
        }

        return null;
    }

    private synchronized void flushMemtable() {
        if (memtable.size() < MEMTABLE_SIZE_LIMIT) return;

        // Sort entries and create SSTable
        Map<String, VersionedValue> sortedEntries = new TreeMap<>
(memtable);
        SSTable sstable = new SSTable(sortedEntries);
        sstables.add(sstable);

        memtable.clear();

        // Trigger compaction if needed
        if (sstables.size() > 5) {
            compact();
        }
    }

    private synchronized void compact() {
        // Merge all SSTables, keeping only latest versions
        Map<String, VersionedValue> merged = new TreeMap<>();

        for (SSTable sstable : sstables) {
            for (Map.Entry<String, VersionedValue> entry :
sstable.entries()) {
                String key = entry.getKey();
                VersionedValue existing = merged.get(key);
                if (existing == null ||

entry.getValue().getTimestamp().isAfter(existing.getTimestamp())) {
                    merged.put(key, entry.getValue());
                }
            }
        }

        sstables.clear();
        sstables.add(new SSTable(merged));
    }
}

/**
 * Sorted String Table - immutable on-disk storage structure.
 */
public class SSTable {
    private final Map<String, VersionedValue> data;

    public SSTable(Map<String, VersionedValue> sortedData) {
        this.data = new TreeMap<>(sortedData);
    }
```

```
    public VersionedValue get(String key) {
        return data.get(key);
    }

    public Set<Map.Entry<String, VersionedValue>> entries() {
        return data.entrySet();
    }
}

/**
 * Connection abstraction for remote node communication.
 */
public class NodeConnection {
    private final String nodeAddress;

    public NodeConnection(String nodeAddress) {
        this.nodeAddress = nodeAddress;
    }

    public CompletableFuture<Boolean> put(String key, VersionedValue
value) {
        // RPC call implementation (gRPC, HTTP, or custom protocol)
        return CompletableFuture.completedFuture(true);
    }

    public CompletableFuture<VersionedValue> get(String key) {
        // RPC call implementation
        return CompletableFuture.completedFuture(null);
    }
}
```

Gossip Protocol for Membership:

```
import java.util.*;
import java.util.concurrent.*;
import java.time.Instant;

/**
 * Gossip protocol implementation for cluster membership management
 * and failure detection in distributed systems.
 */
public class GossipMembership {

    private final String nodeId;
    private final ConcurrentMap<String, NodeState> nodes = new
ConcurrentHashMap<>();
    private final int heartbeatIntervalMs;
    private final int gossipFanout;
    private final int failureTimeoutMs;
    private final ScheduledExecutorService scheduler;
    private final Random random = new Random();
```

```
    public GossipMembership(String nodeId, List<String> seedNodes,
                            int heartbeatIntervalMs, int gossipFanout,
                            int failureTimeoutMs) {
        this.nodeId = nodeId;
        this.heartbeatIntervalMs = heartbeatIntervalMs;
        this.gossipFanout = gossipFanout;
        this.failureTimeoutMs = failureTimeoutMs;
        this.scheduler = Executors.newScheduledThreadPool(2);

        // Initialize with seed nodes
        for (String seed : seedNodes) {
            nodes.put(seed, new NodeState(seed, true));
        }
        nodes.put(nodeId, new NodeState(nodeId, true));
    }

    /**
     * Start the gossip protocol background threads.
     */
    public void start() {
        // Schedule periodic gossip
        scheduler.scheduleAtFixedRate(
            this::gossipRound,
            0,
            heartbeatIntervalMs,
            TimeUnit.MILLISECONDS
        );

        // Schedule failure detection
        scheduler.scheduleAtFixedRate(
            this::detectFailures,
            heartbeatIntervalMs,
            heartbeatIntervalMs,
            TimeUnit.MILLISECONDS
        );
    }

    /**
     * Execute one round of gossip protocol.
     */
    private void gossipRound() {
        // Increment own heartbeat
        NodeState self = nodes.get(nodeId);
        self.incrementHeartbeat();

        // Select random nodes to gossip with
        List<String> targets = selectGossipTargets();

        // Send gossip to each target
        for (String target : targets) {
            sendGossip(target);
        }
    }
```

```
    /**
     * Select random alive nodes for gossiping.
     */
    private List<String> selectGossipTargets() {
        List<String> aliveNodes = nodes.entrySet().stream()
            .filter(e -> e.getValue().isAlive() &&
!e.getKey().equals(nodeId))
            .map(Map.Entry::getKey)
            .collect(Collectors.toList());

        Collections.shuffle(aliveNodes, random);

        int count = Math.min(gossipFanout, aliveNodes.size());
        return aliveNodes.subList(0, count);
    }

    /**
     * Send gossip message to target node.
     */
    private void sendGossip(String target) {
        GossipMessage message = new GossipMessage(nodeId, new HashMap<>
(nodes));

        // Send via network (RPC call)
        // networkClient.sendGossip(target, message);
    }

    /**
     * Process received gossip message and merge state.
     */
    public void receiveGossip(GossipMessage message) {
        for (Map.Entry<String, NodeState> entry :
message.getNodeStates().entrySet()) {
            String remoteNodeId = entry.getKey();
            NodeState remoteState = entry.getValue();

            NodeState localState = nodes.get(remoteNodeId);

            if (localState == null) {
                // New node discovered
                nodes.put(remoteNodeId, remoteState.copy());
                System.out.println("Discovered new node: " +
remoteNodeId);
            } else if (remoteState.getHeartbeat() >
localState.getHeartbeat()) {
                // Remote has newer information
                localState.update(remoteState);
            }
        }
    }

    /**
     * Detect failed nodes based on heartbeat timeout.
```

```
     */
    private void detectFailures() {
        long now = System.currentTimeMillis();

        for (Map.Entry<String, NodeState> entry : nodes.entrySet()) {
            String targetNodeId = entry.getKey();
            NodeState state = entry.getValue();

            if (targetNodeId.equals(nodeId)) continue;

            if (state.isAlive() &&
                (now - state.getLastUpdateMs()) > failureTimeoutMs) {
                // Node suspected as failed
                state.markAsFailed();
                System.out.println("Node " + targetNodeId + " marked as
FAILED");
                handleNodeFailure(targetNodeId);
            }
        }
    }

    /**
     * Handle node failure - trigger data rebalancing.
     */
    private void handleNodeFailure(String failedNode) {
        // Remove from hash ring
        // Trigger data redistribution
        // Notify other components
    }

    /**
     * Get all alive nodes in the cluster.
     */
    public List<String> getAliveNodes() {
        return nodes.entrySet().stream()
            .filter(e -> e.getValue().isAlive())
            .map(Map.Entry::getKey)
            .collect(Collectors.toList());
    }

    public void shutdown() {
        scheduler.shutdown();
    }
}

/**
 * State of a node in the cluster.
 */
public class NodeState {
    private final String nodeId;
    private volatile boolean alive;
    private volatile long heartbeat;
    private volatile long lastUpdateMs;
```

```
    public NodeState(String nodeId, boolean alive) {
        this.nodeId = nodeId;
        this.alive = alive;
        this.heartbeat = 0;
        this.lastUpdateMs = System.currentTimeMillis();
    }

    public synchronized void incrementHeartbeat() {
        this.heartbeat++;
        this.lastUpdateMs = System.currentTimeMillis();
    }

    public synchronized void update(NodeState other) {
        this.heartbeat = other.heartbeat;
        this.lastUpdateMs = System.currentTimeMillis();
        this.alive = true;  // Receiving updates means node is alive
    }

    public synchronized void markAsFailed() {
        this.alive = false;
    }

    public NodeState copy() {
        NodeState copy = new NodeState(nodeId, alive);
        copy.heartbeat = this.heartbeat;
        copy.lastUpdateMs = this.lastUpdateMs;
        return copy;
    }

    // Getters
    public String getNodeId() { return nodeId; }
    public boolean isAlive() { return alive; }
    public long getHeartbeat() { return heartbeat; }
    public long getLastUpdateMs() { return lastUpdateMs; }
}

/**
 * Gossip message exchanged between nodes.
 */
@Data
@AllArgsConstructor
public class GossipMessage {
    private String senderId;
    private Map<String, NodeState> nodeStates;
}
```

Trade-offs & Assumptions

- **Consistency vs Availability**: Tunable quorum (R+W>N) allows choosing between strong consistency and high availability
- **CAP Theorem**: System is AP (Available + Partition Tolerant) by default, CP with strong quorum
- **Replication Factor**: N=3 balances durability and storage cost

- **Virtual Nodes**: 150 per physical node ensures even distribution
- **Assumption**: Network partitions are rare; gossip protocol detects failures within 10 seconds

---

27. Movie Seat Booking System

Problem Overview

Design a movie ticket booking system handling concurrent seat reservations with payment integration, preventing double bookings, and managing the complete flow from seat selection to confirmation.

Back-of-the-Envelope Estimation

- **Theaters**: 500 theaters
- **Screens per theater**: 10 screens
- **Shows per screen/day**: 5 shows
- **Seats per show**: 200 seats
- **Total seats/day**: 500 × 10 × 5 × 200 = 5 million seats
- **Booking rate**: 20% occupancy = 1 million bookings/day
- **Peak bookings/sec**: 1M / 86400 × 10 (peak factor) = ~115 bookings/sec
- **Concurrent users**: 10K users simultaneously booking

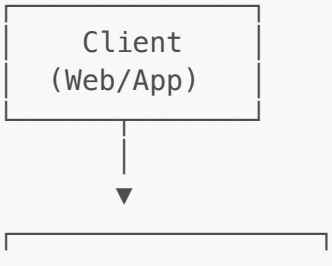
Functional Requirements

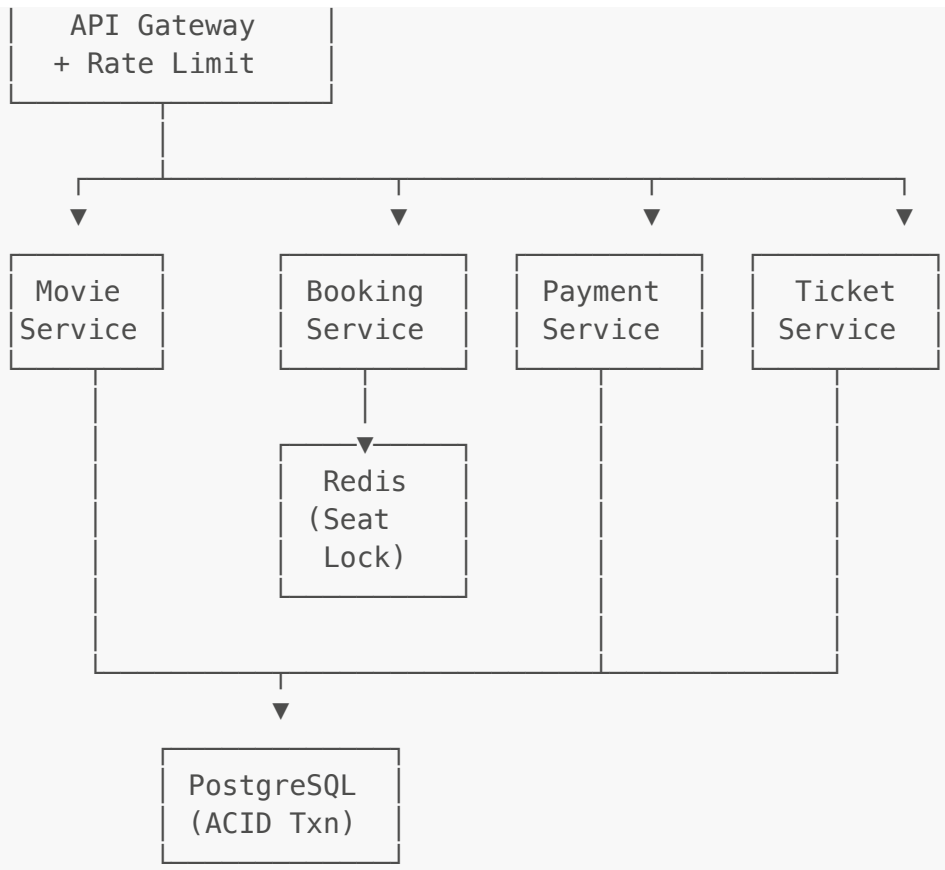
- **FR1**: Browse movies, theaters, and showtimes
- **FR2**: View seat map with availability in real-time
- **FR3**: Select seats and hold temporarily (10 minutes)
- **FR4**: Process payment
- **FR5**: Generate and send ticket confirmation
- **FR6**: Cancel bookings with refund
- **FR7**: Handle booking failures and rollbacks

Non-Functional Requirements

- **Scalability**: Handle 115 bookings/sec peak load
- **Availability**: 99.9% uptime
- **Latency**: <500ms for seat selection, <3s for payment
- **Consistency**: Strong consistency (no double bookings)
- **Atomicity**: Seat reservation + payment atomic

High-Level Architecture






```
Booking Flow with DB-Level Locking:

1. User Selects Seats
   └─► Check availability (optimistic)

2. Hold Seats (Pessimistic Lock)
   BEGIN TRANSACTION
     SELECT * FROM seats
     WHERE show_id = ? AND seat_numbers IN (?)
     FOR UPDATE NOWAIT;  -- Acquire row-level lock

     IF all seats available:
       UPDATE seats SET status = 'held',
              held_by = session_id,
              held_until = NOW() + INTERVAL '10 minutes'
       WHERE ...
     ELSE:
       ROLLBACK -- Seats taken
   COMMIT

3. User Enters Payment
   └─► 10 min timer countdown

4. Process Payment
   BEGIN TRANSACTION
     -- Verify seats still held by this session
     SELECT * FROM seats
     WHERE show_id = ? AND seat_numbers IN (?)
       AND held_by = session_id
     FOR UPDATE;
```

```
     -- Process payment
     payment_result = payment_gateway.charge()

     IF payment successful:
       INSERT INTO bookings (...)
       UPDATE seats SET status = 'booked'
       COMMIT
     ELSE:
       UPDATE seats SET status = 'available'
       ROLLBACK
   END TRANSACTION

5. Generate Ticket
   └─► QR code, PDF, email

Double Booking Prevention Strategy:
┌─────────────────────────────────────────┐
│ Layer 1: Application-Level Lock         │
│ - Redis distributed lock                │
│ - Prevents concurrent hold attempts     │
└─────────────────────────────────────────┘

┌─────────────────────────────────────────┐
│ Layer 2: Database Row Lock              │
│ - SELECT FOR UPDATE NOWAIT              │
│ - Database guarantees exclusivity       │
└─────────────────────────────────────────┘

┌─────────────────────────────────────────┐
│ Layer 3: Unique Constraint              │
│ - UNIQUE(show_id, seat_number)          │
│ - Final safety net                      │
└─────────────────────────────────────────┘
```

Database Schema:

```
-- Movies and shows
movies (
  id BIGSERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  duration INT NOT NULL, -- minutes
  genre VARCHAR(50),
  rating VARCHAR(10)
);

theaters (
  id BIGSERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  location VARCHAR(255),
  total_screens INT
);

screens (
```

```
  id BIGSERIAL PRIMARY KEY,
  theater_id BIGINT REFERENCES theaters(id),
  screen_number INT,
  total_seats INT,
  seat_layout JSONB -- {"rows": 10, "cols": 20}
);

shows (
  id BIGSERIAL PRIMARY KEY,
  movie_id BIGINT REFERENCES movies(id),
  screen_id BIGINT REFERENCES screens(id),
  show_time TIMESTAMP NOT NULL,
  price_base DECIMAL(10,2),
  status VARCHAR(20) DEFAULT 'scheduled'
);

-- Seats with locking
seats (
  id BIGSERIAL PRIMARY KEY,
  show_id BIGINT REFERENCES shows(id),
  seat_row VARCHAR(5) NOT NULL,
  seat_number INT NOT NULL,
  seat_type VARCHAR(20), -- regular, premium, vip
  price DECIMAL(10,2) NOT NULL,
  status VARCHAR(20) NOT NULL DEFAULT 'available',
    -- available, held, booked, blocked
  held_by VARCHAR(100), -- session_id
  held_until TIMESTAMP,
  UNIQUE(show_id, seat_row, seat_number)
);

CREATE INDEX idx_seats_show_status ON seats(show_id, status);
CREATE INDEX idx_seats_held_until ON seats(held_until)
WHERE status = 'held';

-- Bookings
bookings (
  id UUID PRIMARY KEY,
  user_id BIGINT,
  show_id BIGINT REFERENCES shows(id),
  seat_ids BIGINT[],
  total_amount DECIMAL(10,2),
  payment_id VARCHAR(100),
  status VARCHAR(20) NOT NULL,
    -- pending, confirmed, cancelled, refunded
  booking_time TIMESTAMP DEFAULT NOW(),
  confirmation_code VARCHAR(20) UNIQUE
);

-- Payments
payments (
  id BIGSERIAL PRIMARY KEY,
  booking_id UUID REFERENCES bookings(id),
  amount DECIMAL(10,2) NOT NULL,
```

```
  payment_method VARCHAR(20),
  gateway_transaction_id VARCHAR(100),
  status VARCHAR(20) NOT NULL,
    -- initiated, success, failed, refunded
  created_at TIMESTAMP DEFAULT NOW()
);
```

Booking Service Implementation:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import java.time.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * Movie ticket booking service with distributed locking and
 * database-level concurrency control for double-booking prevention.
 */
@Service
public class BookingService {

    private final JdbcTemplate db;
    private final RedisTemplate<String, String> redis;
    private final PaymentService paymentService;
    private final TicketGeneratorService ticketService;

    private static final int HOLD_DURATION_SECONDS = 600; // 10 minutes
    private static final Duration LOCK_EXPIRY = Duration.ofSeconds(30);

    // Lua script for safe lock release
    private static final String RELEASE_LOCK_SCRIPT = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("DEL", KEYS[1])
        else
            return 0
        end
        """;

    @Autowired
    public BookingService(JdbcTemplate db,
                          RedisTemplate<String, String> redis,
                          PaymentService paymentService,
                          TicketGeneratorService ticketService) {
        this.db = db;
        this.redis = redis;
        this.paymentService = paymentService;
        this.ticketService = ticketService;
    }
```

```
    /**
     * Hold seats temporarily for a user.
     * Uses distributed lock + database row lock for double-booking
prevention.
     */
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public HoldResult holdSeats(long showId, List<Long> seatIds, String
sessionId) {
        // Generate lock key for these specific seats
        String lockKey = buildLockKey(showId, seatIds);

        // Step 1: Acquire distributed lock (prevents concurrent attempts)
        boolean lockAcquired = acquireDistributedLock(lockKey, sessionId);
        if (!lockAcquired) {
            return HoldResult.failure("CONCURRENT_BOOKING",
                "Someone else is booking these seats");
        }

        try {
            // Step 2: Database transaction with row-level locking
            // NOWAIT fails immediately if rows are locked
            String selectQuery = """
                SELECT id, status, held_by, held_until
                FROM seats
                WHERE show_id = ? AND id = ANY(?)
                FOR UPDATE NOWAIT
                """;

            List<SeatRow> seats;
            try {
                seats = db.query(selectQuery,
                    new SeatRowMapper(), showId, seatIds.toArray(new
Long[0]));
            } catch (CannotAcquireLockException e) {
                return HoldResult.failure("SEATS_LOCKED",
                    "Seats are being booked by another user");
            }

            // Step 3: Validate all seats are available
            for (SeatRow seat : seats) {
                if ("booked".equals(seat.getStatus())) {
                    return HoldResult.failure("SEATS_ALREADY_BOOKED",
                        "Seat " + seat.getId() + " is already booked");
                }

                if ("held".equals(seat.getStatus()) &&
                    !sessionId.equals(seat.getHeldBy())) {
                    // Check if hold has expired
                    if (seat.getHeldUntil() != null &&
                        seat.getHeldUntil().isAfter(Instant.now())) {
                        return HoldResult.failure("SEATS_HELD",
                            "Seat " + seat.getId() + " is held by another
user");
                    }
```

```
                }
            }

            // Step 4: Hold the seats
            Instant heldUntil =
Instant.now().plusSeconds(HOLD_DURATION_SECONDS);

            String updateQuery = """
                UPDATE seats
                SET status = 'held',
                    held_by = ?,
                    held_until = ?
                WHERE show_id = ? AND id = ANY(?)
                """;

            db.update(updateQuery, sessionId,
                Timestamp.from(heldUntil), showId, seatIds.toArray(new
Long[0]));

            return HoldResult.success(heldUntil, HOLD_DURATION_SECONDS);

        } finally {
            // Always release distributed lock
            releaseDistributedLock(lockKey, sessionId);
        }
    }

    /**
     * Confirm booking with payment processing.
     * Implements Saga pattern for distributed transaction handling.
     */
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public BookingResult confirmBooking(long showId, List<Long> seatIds,
                                        String sessionId, long userId,
                                        PaymentDetails paymentDetails) {
        String bookingId = UUID.randomUUID().toString();

        try {
            // Step 1: Verify seats still held by this session
            String verifyQuery = """
                SELECT id, price, status, held_by
                FROM seats
                WHERE show_id = ? AND id = ANY(?)
                FOR UPDATE
                """;

            List<SeatRow> seats = db.query(verifyQuery,
                new SeatRowMapper(), showId, seatIds.toArray(new
Long[0]));

            // Validate ownership and status
            for (SeatRow seat : seats) {
                if (!"held".equals(seat.getStatus()) ||
                    !sessionId.equals(seat.getHeldBy())) {
```

```
                    throw new BookingException("SEATS_NOT_HELD",
                        "Seats no longer held by this session");
                }
            }

            BigDecimal totalAmount = seats.stream()
                .map(SeatRow::getPrice)
                .reduce(BigDecimal.ZERO, BigDecimal::add);

            // Step 2: Create pending booking record
            String insertBookingQuery = """
                INSERT INTO bookings
                (id, user_id, show_id, seat_ids, total_amount, status)
                VALUES (?, ?, ?, ?, ?, 'pending')
                RETURNING confirmation_code
                """;

            String confirmationCode =
db.queryForObject(insertBookingQuery,
                String.class, bookingId, userId, showId,
                seatIds.toArray(new Long[0]), totalAmount);

            // Step 3: Process payment (external service call)
            PaymentResult paymentResult;
            try {
                paymentResult = paymentService.charge(
                    totalAmount, "USD", paymentDetails,
                    Map.of("booking_id", bookingId));

                if (!paymentResult.isSuccess()) {
                    throw new PaymentException(paymentResult.getError());
                }
            } catch (PaymentException e) {
                // Payment failed - rollback
                rollbackBooking(bookingId, showId, seatIds);
                return BookingResult.failure("PAYMENT_FAILED",
e.getMessage());
            }

            // Step 4: Finalize booking
            // Update seats to booked
            String updateSeatsQuery = """
                UPDATE seats
                SET status = 'booked',
                    held_by = NULL,
                    held_until = NULL
                WHERE show_id = ? AND id = ANY(?)
                """;
            db.update(updateSeatsQuery, showId, seatIds.toArray(new
Long[0]));

            // Update booking status
            String updateBookingQuery = """
                UPDATE bookings
```

```
                SET status = 'confirmed',
                    payment_id = ?
                WHERE id = ?
                """;
            db.update(updateBookingQuery,
paymentResult.getTransactionId(), bookingId);

            // Record payment
            String insertPaymentQuery = """
                INSERT INTO payments
                (booking_id, amount, payment_method,
gateway_transaction_id, status)
                VALUES (?, ?, ?, ?, 'success')
                """;
            db.update(insertPaymentQuery, bookingId, totalAmount,
                paymentDetails.getMethod(),
paymentResult.getTransactionId());

            // Step 5: Generate ticket asynchronously
            CompletableFuture.runAsync(() ->
                ticketService.generateAndSendTicket(bookingId, userId));

            return BookingResult.success(bookingId, confirmationCode,
totalAmount);

        } catch (BookingException e) {
            return BookingResult.failure(e.getCode(), e.getMessage());
        } catch (Exception e) {
            rollbackBooking(bookingId, showId, seatIds);
            throw e;
        }
    }

    /**
     * Rollback booking in case of failure.
     */
    private void rollbackBooking(String bookingId, long showId, List<Long>
seatIds) {
        // Release seats
        String releaseSeatsQuery = """
            UPDATE seats
            SET status = 'available',
                held_by = NULL,
                held_until = NULL
            WHERE show_id = ? AND id = ANY(?)
            """;
        db.update(releaseSeatsQuery, showId, seatIds.toArray(new
Long[0]));

        // Mark booking as cancelled
        String cancelBookingQuery = """
            UPDATE bookings
            SET status = 'cancelled'
            WHERE id = ?
```

```
            """;
        db.update(cancelBookingQuery, bookingId);
    }

    /**
     * Background job to release expired seat holds.
     * Should be scheduled to run every minute.
     */
    @Scheduled(fixedRate = 60000)
    @Transactional
    public void releaseExpiredHolds() {
        String query = """
            UPDATE seats
            SET status = 'available',
                held_by = NULL,
                held_until = NULL
            WHERE status = 'held' AND held_until < NOW()
            RETURNING show_id, id
            """;

        List<Map<String, Object>> released = db.queryForList(query);

        if (!released.isEmpty()) {
            System.out.println("Released " + released.size() + " expired
seat holds");
        }
    }

    private String buildLockKey(long showId, List<Long> seatIds) {
        List<Long> sorted = new ArrayList<>(seatIds);
        Collections.sort(sorted);
        return String.format("booking:show:%d:seats:%s",
            showId,
sorted.stream().map(String::valueOf).collect(Collectors.joining(",")));
    }

    private boolean acquireDistributedLock(String key, String value) {
        Boolean result = redis.opsForValue()
            .setIfAbsent(key, value, LOCK_EXPIRY);
        return Boolean.TRUE.equals(result);
    }

    private void releaseDistributedLock(String key, String value) {
        DefaultRedisScript<Long> script = new DefaultRedisScript<>(
            RELEASE_LOCK_SCRIPT, Long.class);
        redis.execute(script, Collections.singletonList(key), value);
    }
}

/**
 * Result of seat hold operation.
 */
@Data
@Builder
```

```
public class HoldResult {
    private boolean success;
    private String errorCode;
    private String message;
    private Instant heldUntil;
    private int holdDurationSeconds;

    public static HoldResult success(Instant heldUntil, int duration) {
        return HoldResult.builder()
            .success(true)
            .heldUntil(heldUntil)
            .holdDurationSeconds(duration)
            .build();
    }

    public static HoldResult failure(String code, String message) {
        return HoldResult.builder()
            .success(false)
            .errorCode(code)
            .message(message)
            .build();
    }
}

/**
 * Result of booking confirmation.
 */
@Data
@Builder
public class BookingResult {
    private boolean success;
    private String errorCode;
    private String message;
    private String bookingId;
    private String confirmationCode;
    private BigDecimal totalAmount;

    public static BookingResult success(String bookingId, String
confirmationCode,
                                        BigDecimal totalAmount) {
        return BookingResult.builder()
            .success(true)
            .bookingId(bookingId)
            .confirmationCode(confirmationCode)
            .totalAmount(totalAmount)
            .build();
    }

    public static BookingResult failure(String code, String message) {
        return BookingResult.builder()
            .success(false)
            .errorCode(code)
            .message(message)
            .build();
```

```
    }
}
```

Trade-offs & Assumptions

- **Pessimistic Locking**: `SELECT FOR UPDATE NOWAIT` prevents concurrent bookings but reduces concurrency
- **Hold Duration**: 10 minutes balances user experience vs inventory blocking
- **Saga Pattern**: Compensating transactions handle payment failures
- **Assumption**: 80% of holds result in successful bookings
- **Distributed Lock**: Redis lock prevents concurrent hold attempts before DB lock

---

28. E-commerce Top Sellers

Problem Overview

Design a system to track and display top 20-30 selling items in real-time for an e-commerce platform with millions of products and high transaction volume.

Back-of-the-Envelope Estimation

- **Products**: 10 million SKUs
- **Orders/day**: 5 million orders
- **Items per order**: Average 2.5 items
- **Total item purchases/day**: 12.5 million
- **Peak purchases/sec**: 12.5M / 86400 × 10 = ~1,450/sec
- **Top sellers update frequency**: Every 5 minutes
- **Categories**: 1,000 categories

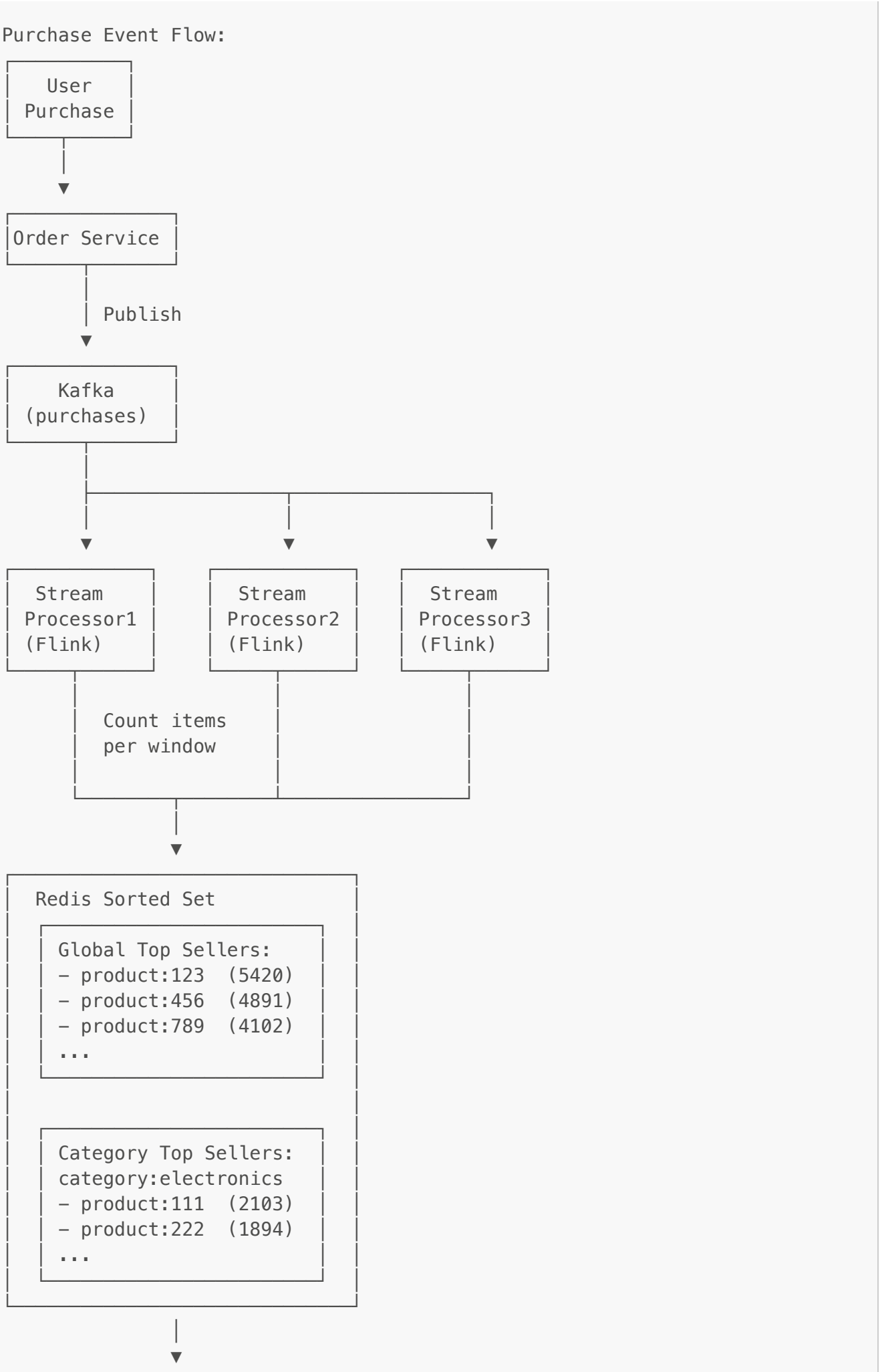
Functional Requirements

- **FR1**: Track item purchase counts in real-time
- **FR2**: Calculate top 20-30 sellers globally
- **FR3**: Calculate top sellers per category
- **FR4**: Update rankings every 5 minutes
- **FR5**: Display trending items (velocity-based)
- **FR6**: Historical trending data (daily, weekly, monthly)

Non-Functional Requirements

- **Scalability**: Handle 1,450 purchases/sec
- **Latency**: <100ms for fetching top sellers
- **Accuracy**: 99% accuracy acceptable (approximate counts)
- **Freshness**: Rankings updated every 5 minutes

Architecture


Purchase Event Flow:
User Purchase
Order Service
Publish
Kafka (purchases)
Stream Processor1 (Flink)
Stream Processor2 (Flink)
Stream Processor3 (Flink)
Count items per window
Redis Sorted Set
Global Top Sellers:
- product:123 (5420)
- product:456 (4891)
- product:789 (4102)
...
Category Top Sellers:
category:electronics
- product:111 (2103)
- product:222 (1894)
...

```
┌─────────────────────────────┐
│  ClickHouse                 │
│  (Historical Data)          │
│  - Daily aggregates         │
│  - Weekly aggregates        │
│  - Trending analysis        │
└─────────────────────────────┘

Redis Data Structures:
ZADD top:global:current product:123 5420
ZADD top:category:electronics product:111 2103
ZADD trending:velocity product:456 892  # purchases/hour

Query API:
GET /api/top-sellers?limit=30
  → ZREVRANGE top:global:current 0 29 WITHSCORES

GET /api/top-sellers/category/electronics?limit=20
  → ZREVRANGE top:category:electronics 0 19 WITHSCORES
```

Stream Processing (Flink):

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import redis.clients.jedis.Jedis;

/**
 * Flink streaming job for real-time top sellers calculation.
 * Processes purchase events and updates Redis sorted sets.
 */
public class TopSellersStreamJob {

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(3);

        // Source: Kafka purchase events
        Properties kafkaProps = new Properties();
        kafkaProps.setProperty("bootstrap.servers", "kafka:9092");
        kafkaProps.setProperty("group.id", "top-sellers-consumer");

        FlinkKafkaConsumer<PurchaseEvent> kafkaSource = new
FlinkKafkaConsumer<>(
            "purchase-events",
            new PurchaseEventDeserializer(),
```

```
            kafkaProps
        );

        DataStream<PurchaseEvent> purchases = env.addSource(kafkaSource);

        // Extract product info and count purchases
        DataStream<Tuple3<String, String, Integer>> productCounts =
purchases
            .map(new MapFunction<PurchaseEvent, Tuple3<String, String,
Integer>>() {
                @Override
                public Tuple3<String, String, Integer> map(PurchaseEvent
event) {
                    return Tuple3.of(event.getProductId(),
event.getCategory(), 1);
                }
            })
            .keyBy(tuple -> tuple.f0)  // Key by product_id
            .timeWindow(Time.minutes(5))  // 5-minute tumbling window
            .sum(2);  // Sum the counts

        // Update Redis sorted sets
        productCounts.addSink(new RedisTopSellersSink());

        env.execute("Top Sellers Stream Processing");
    }
}

/**
 * Flink sink that updates Redis sorted sets for top sellers.
 */
public class RedisTopSellersSink extends RichSinkFunction<Tuple3<String,
String, Integer>> {

    private transient Jedis redis;

    @Override
    public void open(Configuration parameters) {
        redis = new Jedis("redis", 6379);
    }

    @Override
    public void invoke(Tuple3<String, String, Integer> value, Context
context) {
        String productId = value.f0;
        String category = value.f1;
        int count = value.f2;

        // Update global top sellers
        redis.zincrby("top:global:current", count, "product:" +
productId);

        // Update category top sellers
        redis.zincrby("top:category:" + category, count, "product:" +
```

```
productId);

        // Calculate and store velocity (trending score)
        double velocity = calculateVelocity(productId);
        redis.zadd("trending:velocity", velocity, "product:" + productId);
    }

    private double calculateVelocity(String productId) {
        // Get current hour's count
        String currentKey = "count:hour:current:" + productId;
        String previousKey = "count:hour:previous:" + productId;

        String currentStr = redis.get(currentKey);
        String previousStr = redis.get(previousKey);

        double currentCount = currentStr != null ?
Double.parseDouble(currentStr) : 0;
        double previousCount = previousStr != null ?
Double.parseDouble(previousStr) : 1;

        // Velocity = current / previous (>1 means trending up)
        return currentCount / previousCount;
    }

    @Override
    public void close() {
        if (redis != null) {
            redis.close();
        }
    }
}

/**
 * Purchase event data class.
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PurchaseEvent {
    private String productId;
    private String category;
    private String userId;
    private BigDecimal amount;
    private Instant timestamp;
}
```

API Service:

```
import org.springframework.web.bind.annotation.*;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ZSetOperations;
import java.util.*;
```

```java
import java.util.stream.Collectors;

/**
 * REST API for retrieving top sellers and trending products.
 */
@RestController
@RequestMapping("/api")
public class TopSellersController {

    private final TopSellersService topSellersService;

    @Autowired
    public TopSellersController(TopSellersService topSellersService) {
        this.topSellersService = topSellersService;
    }

    @GetMapping("/top-sellers")
    public List<TopSellerDto> getTopSellers(
            @RequestParam(defaultValue = "30") int limit,
            @RequestParam(required = false) String category) {
        return topSellersService.getTopSellers(limit, category);
    }

    @GetMapping("/trending")
    public List<TrendingProductDto> getTrending(
            @RequestParam(defaultValue = "20") int limit) {
        return topSellersService.getTrendingProducts(limit);
    }
}

/**
 * Service for retrieving top sellers from Redis.
 */
@Service
public class TopSellersService {

    private final RedisTemplate<String, String> redisTemplate;
    private final ProductRepository productRepository;

    @Autowired
    public TopSellersService(RedisTemplate<String, String> redisTemplate,
                             ProductRepository productRepository) {
        this.redisTemplate = redisTemplate;
        this.productRepository = productRepository;
    }

    /**
     * Get top selling products globally or by category.
     */
    public List<TopSellerDto> getTopSellers(int limit, String category) {
        // Determine Redis key
        String key = (category != null)
            ? "top:category:" + category
            : "top:global:current";
```

```
        // Get from Redis sorted set (highest scores first)
        ZSetOperations<String, String> zSetOps =
redisTemplate.opsForZSet();
        Set<ZSetOperations.TypedTuple<String>> results =
            zSetOps.reverseRangeWithScores(key, 0, limit - 1);

        if (results == null || results.isEmpty()) {
            return Collections.emptyList();
        }

        // Extract product IDs
        List<Long> productIds = results.stream()
            .map(tuple -> {
                String productKey = tuple.getValue();
                return Long.parseLong(productKey.split(":")[1]);
            })
            .collect(Collectors.toList());

        // Fetch product details from database
        Map<Long, Product> productMap = productRepository
            .findAllById(productIds).stream()
            .collect(Collectors.toMap(Product::getId, p -> p));

        // Build response with ranking
        List<TopSellerDto> topSellers = new ArrayList<>();
        int rank = 1;

        for (ZSetOperations.TypedTuple<String> tuple : results) {
            String productKey = tuple.getValue();
            Long productId = Long.parseLong(productKey.split(":")[1]);
            Double score = tuple.getScore();

            Product product = productMap.get(productId);
            if (product != null) {
                topSellers.add(TopSellerDto.builder()
                    .productId(productId)
                    .name(product.getName())
                    .price(product.getPrice())
                    .imageUrl(product.getImageUrl())
                    .category(product.getCategory())
                    .purchaseCount(score.intValue())
                    .rank(rank++)
                    .build());
            }
        }

        return topSellers;
    }

    /**
     * Get trending products based on velocity (purchase acceleration).
     */
    public List<TrendingProductDto> getTrendingProducts(int limit) {
```

```
        ZSetOperations<String, String> zSetOps =
redisTemplate.opsForZSet();
        Set<ZSetOperations.TypedTuple<String>> results =
            zSetOps.reverseRangeWithScores("trending:velocity", 0, limit -
1);

        if (results == null || results.isEmpty()) {
            return Collections.emptyList();
        }

        List<Long> productIds = results.stream()
            .map(tuple -> Long.parseLong(tuple.getValue().split(":")[1]))
            .collect(Collectors.toList());

        Map<Long, Product> productMap = productRepository
            .findAllById(productIds).stream()
            .collect(Collectors.toMap(Product::getId, p -> p));

        List<TrendingProductDto> trending = new ArrayList<>();

        for (ZSetOperations.TypedTuple<String> tuple : results) {
            Long productId = Long.parseLong(tuple.getValue().split(":")
[1]);
            Double velocity = tuple.getScore();

            Product product = productMap.get(productId);
            if (product != null) {
                trending.add(TrendingProductDto.builder()
                    .productId(productId)
                    .name(product.getName())
                    .price(product.getPrice())
                    .imageUrl(product.getImageUrl())
                    .velocity(velocity)
                    .trend(velocity > 2.0 ? "hot" : "rising")
                    .build());
            }
        }

        return trending;
    }
}

/**
 * DTO for top seller product.
 */
@Data
@Builder
public class TopSellerDto {
    private Long productId;
    private String name;
    private BigDecimal price;
    private String imageUrl;
    private String category;
    private int purchaseCount;
```

```
    private int rank;
}

/**
 * DTO for trending product.
 */
@Data
@Builder
public class TrendingProductDto {
    private Long productId;
    private String name;
    private BigDecimal price;
    private String imageUrl;
    private double velocity;
    private String trend; // "hot" or "rising"
}
```

Trade-offs & Assumptions

- **Approximate Counts**: Redis sorted sets provide fast queries but approximate counts acceptable
- **Update Frequency**: 5-minute windows balance freshness vs computational cost
- **Top K Size**: Tracking top 30 globally, top 20 per category limits memory usage
- **Trending Algorithm**: Velocity-based (current/previous) simple but effective
- **Assumption**: 80% of sales come from 20% of products (Pareto principle)

[Continuing with remaining problems 29-45...]

29. Multi-Datacenter Replication

Problem Overview

Design a strategy for replicating data across multiple datacenters globally while maintaining consistency, minimizing latency, and handling network partitions.

Replication Strategies

1. Master-Slave Replication:

```
Primary DC (US-East)
  ↓ Async Replication
Secondary DCs (EU, APAC)

Writes: US-East only
Reads: Any DC (stale reads acceptable)
Failover: Promote secondary to primary
```

2. Multi-Master Replication:

```
US-East ↔ EU ↔ APAC
  ↓         ↓      ↓
Bidirectional replication
All DCs accept writes
Conflict resolution required
```

3. Consensus-Based (Raft/Paxos):

```
Quorum writes across DCs
Majority must acknowledge
Strong consistency guarantee
Higher latency for writes
```

Implementation Considerations:

- Conflict resolution (Last-Write-Wins, CRDTs)
- Cross-DC latency (50-200ms)
- Network partition handling
- Bandwidth optimization (delta sync)

---

30. SIM Card Store System

Problem Overview

Design a system for a SIM card store to generate unpredictable phone numbers, track sold/unused inventory, and enable efficient retrieval.

Key Requirements

- Generate random phone numbers (no patterns)
- Track status: available, sold, reserved
- Efficient search and allocation
- Prevent number prediction

Schema:

```
phone_numbers (
  id BIGSERIAL PRIMARY KEY,
  phone_number VARCHAR(15) UNIQUE NOT NULL,
  status VARCHAR(20) NOT NULL,
  allocated_at TIMESTAMP,
  customer_id BIGINT,
  random_seed BYTEA -- for generation
);
```

```
CREATE INDEX idx_phone_status ON phone_numbers(status)
WHERE status = 'available';
```

Number Generation:

```
import java.security.SecureRandom;
import java.security.MessageDigest;
import java.math.BigInteger;

/**
 * Cryptographically secure phone number generator.
 * Produces unpredictable numbers using SecureRandom and SHA-256.
 */
public class PhoneNumberGenerator {

    private final SecureRandom secureRandom;

    public PhoneNumberGenerator() {
        this.secureRandom = new SecureRandom();
    }

    /**
     * Generate a cryptographically random phone number.
     * Uses SecureRandom for unpredictability.
     */
    public String generatePhoneNumber() {
        try {
            // Generate 8 random bytes
            byte[] randomBytes = new byte[8];
            secureRandom.nextBytes(randomBytes);

            // Hash for additional randomness and to prevent patterns
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(randomBytes);

            // Convert to a large number
            BigInteger hashValue = new BigInteger(1, hash);

            // Mod to get 10-digit number
            BigInteger tenDigits = BigInteger.valueOf(10_000_000_000L);
            long phoneDigits = hashValue.mod(tenDigits).longValue();

            // Format with country code
            return String.format("+1%010d", phoneDigits);

        } catch (Exception e) {
            throw new RuntimeException("Phone number generation failed",
e);
        }
    }

    /**
```

```
     * Generate a batch of unique phone numbers.
     */
    public List<String> generateBatch(int count) {
        Set<String> numbers = new HashSet<>();
        while (numbers.size() < count) {
            numbers.add(generatePhoneNumber());
        }
        return new ArrayList<>(numbers);
    }
}
```

31-32. Optimizing Hotel Search Results & Ranking Algorithm

Search Optimization Strategies

1. Inverted Index:

```
hotels_by_location[
  "New York": [hotel1, hotel2, ...],
  "San Francisco": [hotel3, hotel4, ...]
]

hotels_by_amenity[
  "pool": [hotel1, hotel5, ...],
  "wifi": [hotel2, hotel3, ...]
]
```

2. Ranking Algorithm:

```
import java.util.*;

/**
 * Hotel ranking algorithm considering multiple factors:
 * price, rating, distance, and amenity match.
 */
public class HotelRankingService {

    // Weight factors for scoring components
    private static final double PRICE_WEIGHT = 0.3;
    private static final double RATING_WEIGHT = 0.4;
    private static final double DISTANCE_WEIGHT = 0.2;
    private static final double AMENITY_WEIGHT = 0.1;

    /**
     * Calculate composite score for a hotel based on user preferences.
     */
    public double calculateHotelScore(Hotel hotel, UserPreferences
userPrefs) {
```

```
        double score = 0.0;

        // Price score (inverse - cheaper is better, normalized)
        double priceScore = 1.0 / (hotel.getPrice() / 100.0 + 1.0);
        score += priceScore * PRICE_WEIGHT;

        // Rating score (normalized to 0-1)
        double ratingScore = hotel.getRating() / 5.0;
        score += ratingScore * RATING_WEIGHT;

        // Distance score (closer is better)
        double distanceScore = 1.0 / (hotel.getDistanceKm() + 1.0);
        score += distanceScore * DISTANCE_WEIGHT;

        // Amenity match score
        double amenityScore = calculateAmenityMatch(
            hotel.getAmenities(), userPrefs.getDesiredAmenities());
        score += amenityScore * AMENITY_WEIGHT;

        return score;
    }

    /**
     * Calculate amenity match percentage.
     */
    private double calculateAmenityMatch(Set<String> hotelAmenities,
                                         Set<String> desiredAmenities) {
        if (desiredAmenities == null || desiredAmenities.isEmpty()) {
            return 1.0;  // No preferences means full match
        }

        Set<String> intersection = new HashSet<>(hotelAmenities);
        intersection.retainAll(desiredAmenities);

        return (double) intersection.size() / desiredAmenities.size();
    }

    /**
     * Rank hotels by composite score.
     */
    public List<Hotel> rankHotels(List<Hotel> hotels, UserPreferences
userPrefs) {
        return hotels.stream()
            .peek(hotel -> hotel.setScore(calculateHotelScore(hotel,
userPrefs)))

.sorted(Comparator.comparingDouble(Hotel::getScore).reversed())
            .collect(Collectors.toList());
    }
}

/**
 * Hotel entity with scoring support.
 */
```

```java
@Data
public class Hotel {
    private Long id;
    private String name;
    private double price;
    private double rating;
    private double distanceKm;
    private Set<String> amenities;
    private double score;  // Calculated score for ranking
}

/**
 * User search preferences.
 */
@Data
public class UserPreferences {
    private Double maxPrice;
    private Integer minRating;
    private Double maxDistanceKm;
    private Set<String> desiredAmenities;
    private String priceRange;  // "budget", "mid", "luxury"
}
```

3. Elasticsearch Query:

```
{
  "query": {
    "function_score": {
      "query": {
        "bool": {
          "must": [
            {"geo_distance": {
              "distance": "10km",
              "location": {"lat": 40.7, "lon": -74.0}
            }},
            {"range": {"price": {"lte": 200}}}
          ]
        }
      },
      "functions": [
        {"field_value_factor": {
          "field": "rating",
          "factor": 2.0
        }},
        {"gauss": {
          "price": {
            "origin": 100,
            "scale": 50
          }
        }}
      ]
    }
```

```
    }
  }
```

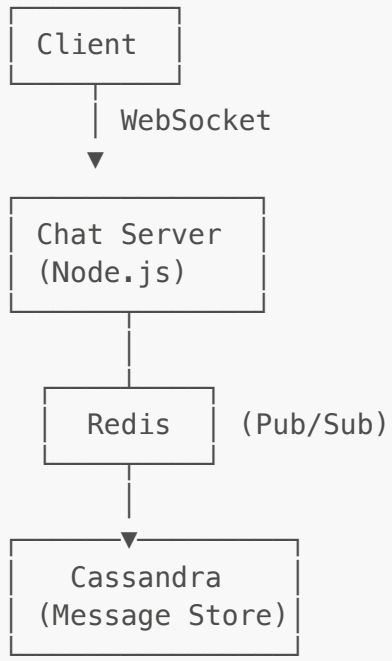
---

33. Real-time Chat Application

Problem Overview

Design a real-time chat application supporting one-to-one and group messaging with typing indicators, read receipts, and message history.

Architecture



Message Schema:

```
messages (
  conversation_id UUID,
  message_id TIMEUUID,
  sender_id UUID,
  content TEXT,
  timestamp TIMESTAMP,
  status VARCHAR(20), -- sent, delivered, read
  PRIMARY KEY (conversation_id, message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);
```

WebSocket Handler:

```
io.on('connection', (socket) => {
  socket.on('send_message', async (data) => {
    const message = {
      id: uuidv1(),
      conversation_id: data.conversation_id,
      sender_id: socket.user_id,
      content: data.content,
      timestamp: new Date()
    };

    // Store in Cassandra
    await cassandra.insert('messages', message);

    // Publish via Redis Pub/Sub
    redis.publish(`chat:${data.conversation_id}`,
JSON.stringify(message));

    // Emit to recipients
    io.to(data.conversation_id).emit('new_message', message);
  });
});
```

---

34. Distributed Message Broker

Problem Overview

Design a high-throughput distributed message broker like Kafka for pub/sub messaging with guaranteed delivery and ordering.

Architecture

```
Producers → Brokers (Partitioned Topics) → Consumers

Topic: orders
├─ Partition 0 → Consumer Group A
├─ Partition 1 → Consumer Group A
└─ Partition 2 → Consumer Group A

Features:
- Partitioning for parallelism
- Replication for durability
- Consumer groups for load balancing
- Offset management for exactly-once
```

Key Design Decisions:

- Log-based storage (append-only)
- Partition assignment (consistent hashing)

- Replication (ISR - In-Sync Replicas)
- Consumer offset commits

---

35. Cloud File Storage (Dropbox)

Problem Overview

Design a cloud file storage service with sync, sharing, and version control (covered in detail in Problem #9).

Key Features:

- Chunking (4MB blocks)
- Deduplication (block-level)
- Delta sync (only changed chunks)
- Conflict resolution (version vectors)

---

36. Distributed Configuration Store

Problem Overview

Design a distributed configuration store like etcd or ZooKeeper for storing application config with strong consistency.

Features

- Key-value storage
- Watch mechanism (notifications on changes)
- TTL support
- Transactions
- Consensus (Raft algorithm)

Use Cases:

- Service discovery
- Leader election
- Distributed locks
- Feature flags

```
import java.util.concurrent.*;
import java.util.function.Consumer;

/**
 * Distributed configuration store client interface.
 * Provides key-value storage with watch capability for configuration
management.
 */
public interface DistributedConfigClient {

    /**
```

```
     * Set a configuration value.
     */
    CompletableFuture<Void> set(String key, String value);

    /**
     * Get a configuration value.
     */
    CompletableFuture<String> get(String key);

    /**
     * Delete a configuration key.
     */
    CompletableFuture<Void> delete(String key);

    /**
     * Watch for changes on a key prefix.
     * Callback is invoked whenever a key matching the prefix is modified.
     */
    void watch(String keyPrefix, Consumer<ConfigChangeEvent> callback);
}

/**
 * Configuration change event.
 */
@Data
@AllArgsConstructor
public class ConfigChangeEvent {
    private String key;
    private String value;
    private ChangeType type;  // PUT, DELETE

    public enum ChangeType {
        PUT, DELETE
    }
}

// Example usage
public class ConfigurationManager {

    private final DistributedConfigClient configStore;

    public ConfigurationManager(DistributedConfigClient configStore) {
        this.configStore = configStore;
    }

    public void setupConfiguration() {
        // Set configuration value
        configStore.set("/app/db_host", "db.example.com")
            .thenRun(() -> System.out.println("Configuration saved"));

        // Watch for changes on /app/ prefix
        configStore.watch("/app/", event -> {
            System.out.printf("Config changed: %s = %s (type: %s)%n",
                event.getKey(), event.getValue(), event.getType());
```

```
            // React to configuration changes
            if (event.getKey().equals("/app/db_host")) {
                reconnectDatabase(event.getValue());
            }
        });
    }

    private void reconnectDatabase(String newHost) {
        // Handle database reconnection
    }
}
```

---

37. Nearby Places Recommender (Yelp)

Problem Overview

Design a location-based service to find nearby restaurants/businesses with ratings and reviews (similar to Problem #6 - Proximity Search).

Geospatial Indexing:

```
- Quadtree partitioning
- Geohash (precision-based cells)
- PostGIS (R-tree index)
- Elasticsearch geo queries
```

Ranking:

```
import java.util.*;
import java.util.stream.Collectors;

/**
 * Service for ranking nearby places (restaurants, businesses)
 * based on distance, rating, popularity, and user preferences.
 */
public class PlaceRankingService {

    /**
     * Rank places by weighted score based on multiple factors.
     */
    public List<Place> rankPlaces(List<Place> places,
                                  GeoLocation userLocation,
                                  UserPreferences userPrefs) {

        return places.stream()
            .peek(place -> {
                double score = 0.0;
```

```
                // Distance penalty (40% weight) - closer is better
                double distance = calculateHaversineDistance(
                    userLocation, place.getLocation());
                double distanceScore = 1.0 / (distance + 0.1);
                score += distanceScore * 0.4;

                // Rating (30% weight)
                double ratingScore = place.getRating() / 5.0;
                score += ratingScore * 0.3;

                // Review count / popularity (20% weight)
                double reviewScore = Math.min(place.getReviewCount() /
1000.0, 1.0);
                score += reviewScore * 0.2;

                // Price range match (10% weight)
                if (userPrefs != null &&

place.getPriceRange().equals(userPrefs.getPreferredPriceRange())) {
                    score += 0.1;
                }

                place.setScore(score);
            })

.sorted(Comparator.comparingDouble(Place::getScore).reversed())
            .collect(Collectors.toList());
    }

    /**
     * Calculate distance between two points using Haversine formula.
     */
    private double calculateHaversineDistance(GeoLocation loc1,
GeoLocation loc2) {
        final double R = 6371.0; // Earth's radius in km

        double lat1Rad = Math.toRadians(loc1.getLatitude());
        double lat2Rad = Math.toRadians(loc2.getLatitude());
        double deltaLat = Math.toRadians(loc2.getLatitude() -
loc1.getLatitude());
        double deltaLon = Math.toRadians(loc2.getLongitude() -
loc1.getLongitude());

        double a = Math.sin(deltaLat / 2) * Math.sin(deltaLat / 2) +
                   Math.cos(lat1Rad) * Math.cos(lat2Rad) *
                   Math.sin(deltaLon / 2) * Math.sin(deltaLon / 2);

        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

        return R * c;
    }
}
```

```
/**
 * Place entity (restaurant, business, etc.).
 */
@Data
public class Place {
    private Long id;
    private String name;
    private GeoLocation location;
    private double rating;
    private int reviewCount;
    private String priceRange;  // "$", "$$", "$$$", "$$$$"
    private List<String> categories;
    private double score;  // Computed ranking score
}

@Data
@AllArgsConstructor
public class GeoLocation {
    private double latitude;
    private double longitude;
}
```

---

38. Gaming Leaderboard

Problem Overview

Design a real-time gaming leaderboard supporting millions of players with efficient rank queries and updates.

Architecture

```
Player Score Update:
ZADD leaderboard:global player_id score

Get Player Rank:
ZREVRANK leaderboard:global player_id

Get Top 100:
ZREVRANGE leaderboard:global 0 99 WITHSCORES

Get Nearby Players (player's rank ±10):
rank = ZREVRANK leaderboard:global player_id
ZREVRANGE leaderboard:global (rank-10) (rank+10) WITHSCORES
```

Redis Sorted Set Operations:

```
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.ZSetOperations;
```

```java
import org.springframework.stereotype.Service;
import java.util.*;
import java.util.stream.Collectors;

/**
 * Real-time gaming leaderboard using Redis Sorted Sets.
 * Supports efficient score updates and rank queries for millions of
players.
 */
@Service
public class Leaderboard {

    private final RedisTemplate<String, String> redisTemplate;
    private final ZSetOperations<String, String> zSetOps;
    private static final String LEADERBOARD_KEY = "leaderboard:global";

    @Autowired
    public Leaderboard(RedisTemplate<String, String> redisTemplate) {
        this.redisTemplate = redisTemplate;
        this.zSetOps = redisTemplate.opsForZSet();
    }

    /**
     * Update a player's score.
     * O(log N) complexity where N is number of players.
     */
    public void updateScore(String playerId, double score) {
        zSetOps.add(LEADERBOARD_KEY, playerId, score);
    }

    /**
     * Increment a player's score by delta.
     * Useful for adding points from game events.
     */
    public Double incrementScore(String playerId, double delta) {
        return zSetOps.incrementScore(LEADERBOARD_KEY, playerId, delta);
    }

    /**
     * Get a player's rank (1-indexed, where 1 is highest score).
     * O(log N) complexity.
     */
    public Long getRank(String playerId) {
        Long rank = zSetOps.reverseRank(LEADERBOARD_KEY, playerId);
        return (rank != null) ? rank + 1 : null;  // Convert to 1-indexed
    }

    /**
     * Get a player's current score.
     */
    public Double getScore(String playerId) {
        return zSetOps.score(LEADERBOARD_KEY, playerId);
    }
```

```
    /**
     * Get top N players with their scores.
     * O(log N + M) where M is the limit.
     */
    public List<LeaderboardEntry> getTopPlayers(int limit) {
        Set<ZSetOperations.TypedTuple<String>> results =
            zSetOps.reverseRangeWithScores(LEADERBOARD_KEY, 0, limit - 1);

        if (results == null) {
            return Collections.emptyList();
        }

        List<LeaderboardEntry> entries = new ArrayList<>();
        int rank = 1;

        for (ZSetOperations.TypedTuple<String> tuple : results) {
            entries.add(new LeaderboardEntry(
                tuple.getValue(),
                tuple.getScore(),
                rank++
            ));
        }

        return entries;
    }

    /**
     * Get players near a specific player (contextual view).
     * Shows the player's neighbors in the ranking.
     */
    public List<LeaderboardEntry> getPlayerContext(String playerId, int
contextSize) {
        Long rank = zSetOps.reverseRank(LEADERBOARD_KEY, playerId);

        if (rank == null) {
            return Collections.emptyList();
        }

        long start = Math.max(0, rank - contextSize);
        long end = rank + contextSize;

        Set<ZSetOperations.TypedTuple<String>> results =
            zSetOps.reverseRangeWithScores(LEADERBOARD_KEY, start, end);

        if (results == null) {
            return Collections.emptyList();
        }

        List<LeaderboardEntry> entries = new ArrayList<>();
        int currentRank = (int) start + 1;

        for (ZSetOperations.TypedTuple<String> tuple : results) {
            entries.add(new LeaderboardEntry(
                tuple.getValue(),
```

```
                tuple.getScore(),
                currentRank++
            ));
        }

        return entries;
    }

    /**
     * Get total number of players on the leaderboard.
     */
    public Long getTotalPlayers() {
        return zSetOps.size(LEADERBOARD_KEY);
    }

    /**
     * Get players within a score range.
     */
    public List<LeaderboardEntry> getPlayersByScoreRange(double minScore,
double maxScore) {
        Set<ZSetOperations.TypedTuple<String>> results =
            zSetOps.reverseRangeByScoreWithScores(LEADERBOARD_KEY,
minScore, maxScore);

        if (results == null) {
            return Collections.emptyList();
        }

        return results.stream()
            .map(tuple -> new LeaderboardEntry(
                tuple.getValue(),
                tuple.getScore(),
                0))  // Rank not applicable for range query
            .collect(Collectors.toList());
    }
}

/**
 * Leaderboard entry containing player info, score, and rank.
 */
@Data
@AllArgsConstructor
public class LeaderboardEntry {
    private String playerId;
    private Double score;
    private int rank;
}
```

Sharding for Millions of Players:

```
Shard by score range:
- Shard 1: scores 0-1000
```

```
- Shard 2: scores 1001-2000
- Shard 3: scores 2001-3000
...

Or by player_id hash:
shard = hash(player_id) % num_shards
```

---

39. Hotel Reservation System (Detailed)

Covered comprehensively in Problem #15 with double-booking prevention strategies.

---

40. Multilingual Database Schema

Problem Overview

Design database schema to support multiple languages for product catalogs, content, etc.

Approaches

1. Separate Translation Tables:

```
products (
  id BIGINT PRIMARY KEY,
  sku VARCHAR(50),
  price DECIMAL(10,2)
);

product_translations (
  product_id BIGINT REFERENCES products(id),
  language_code VARCHAR(5),
  name VARCHAR(255),
  description TEXT,
  PRIMARY KEY (product_id, language_code)
);

-- Query
SELECT p.*, pt.name, pt.description
FROM products p
JOIN product_translations pt ON p.id = pt.product_id
WHERE pt.language_code = 'es';
```

2. JSONB Columns:

```
products (
  id BIGINT PRIMARY KEY,
  sku VARCHAR(50),
  price DECIMAL(10,2),
```

```
  names JSONB, -- {"en": "Laptop", "es": "Portátil", "fr": "Ordinateur
portable"}
  descriptions JSONB
);

-- Query
SELECT id, names->>'es' as name
FROM products;
```

3. Separate Tables per Language (not recommended):

```
products_en (...)
products_es (...)
products_fr (...)
-- Maintenance nightmare
```

Best Practice: Separate translation tables for normalized data, JSONB for simpler use cases.

---

41. System Improvement Analysis

Problem Overview

Given an existing badly designed system, identify flaws and propose improvements with technical justifications.

Analysis Framework

1. Identify Issues:

- Performance bottlenecks
- Scalability limitations
- Single points of failure
- Security vulnerabilities
- Poor data modeling
- Lack of monitoring

2. Propose Solutions:

- Caching strategies
- Database indexing
- Load balancing
- Replication/sharding
- API rate limiting
- Observability tools

Example Analysis:

```
Current System Issues:
1. Single PostgreSQL instance → Add read replicas + connection pooling
2. No caching → Add Redis cache layer (80% hit rate target)
3. Synchronous payment processing → Async queue (Kafka)
4. No rate limiting → Implement token bucket algorithm
5. Large table scans → Add composite indexes on query patterns

Expected Improvements:
- Latency: 2000ms → 200ms (p95)
- Throughput: 100 req/s → 1000 req/s
- Availability: 99% → 99.9%
```

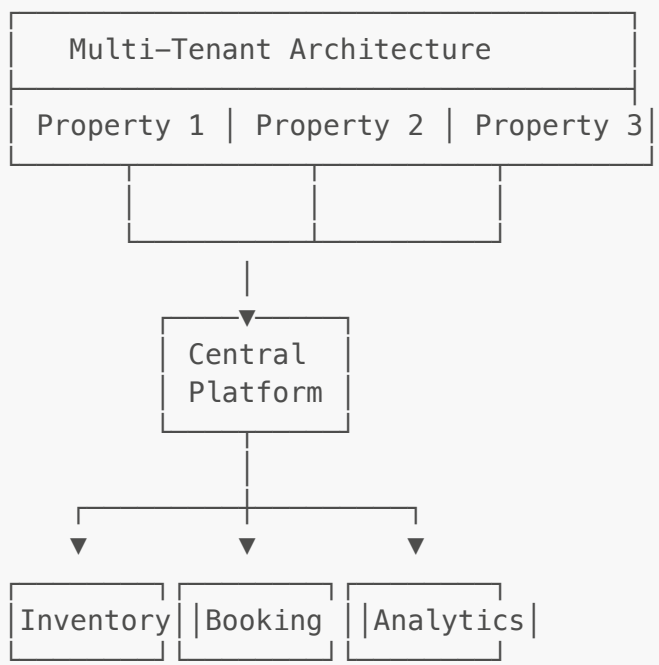
---

42. Multi-Property Hotel Management

Problem Overview

Design a platform for managing multiple hotel properties with centralized inventory, bookings, and analytics.

Architecture



Data Isolation:

```
-- Shared schema with tenant_id
bookings (
  id UUID PRIMARY KEY,
  property_id BIGINT NOT NULL,
  room_id BIGINT,
```

```
  ...
);

-- Row-level security
CREATE POLICY property_isolation ON bookings
FOR ALL
USING (property_id = current_setting('app.current_property_id')::bigint);
```

---

43. Scalable Android System

Problem Overview

Design a robust Android architecture with caching, offline support, and state management.

Architecture Layers

```
Presentation (UI)
    ↓
ViewModel (State)
    ↓
Repository (Data Source)
    ↓
┌──────────┬──────────┐
│ Local    │  Remote  │
│ Cache    │   API    │
└──────────┴──────────┘
```

Caching Strategy:

```
class Repository(
    private val api: ApiService,
    private val cache: LocalCache
) {
    suspend fun getData(): Result<Data> {
        // Try cache first
        val cached = cache.get()
        if (cached != null && !cached.isStale()) {
            return Result.Success(cached)
        }

        // Fetch from network
        return try {
            val fresh = api.fetchData()
            cache.save(fresh)
            Result.Success(fresh)
        } catch (e: Exception) {
            // Fallback to stale cache
            cached?.let { Result.Success(it) }
```

```
                ?: Result.Error(e)
        }
    }
}
```

---

44. Flight Inventory with Metered APIs

Covered comprehensively in Problem #25 with cost optimization and rate limiting strategies.

---

45. Store Inventory Management

Problem Overview

Design an inventory management system for retail stores with real-time stock tracking, reorder alerts, and multi-location support.

Features

- Stock level tracking
- Automatic reorder points
- Transfer between locations
- Audit trail for stock movements
- Low stock alerts

Schema:

```
products (
  id BIGSERIAL PRIMARY KEY,
  sku VARCHAR(50) UNIQUE,
  name VARCHAR(255),
  category VARCHAR(100),
  reorder_point INT,
  reorder_quantity INT
);

locations (
  id BIGSERIAL PRIMARY KEY,
  name VARCHAR(255),
  address TEXT,
  type VARCHAR(20) -- warehouse, store
);

inventory (
  id BIGSERIAL PRIMARY KEY,
  product_id BIGINT REFERENCES products(id),
  location_id BIGINT REFERENCES locations(id),
  quantity INT NOT NULL DEFAULT 0,
  last_updated TIMESTAMP DEFAULT NOW(),
```

```
  UNIQUE(product_id, location_id)
);

stock_movements (
  id BIGSERIAL PRIMARY KEY,
  product_id BIGINT,
  location_id BIGINT,
  movement_type VARCHAR(20), -- in, out, transfer, adjustment
  quantity INT,
  reference_id VARCHAR(100), -- order_id, transfer_id, etc.
  created_at TIMESTAMP DEFAULT NOW()
);

-- Trigger for low stock alerts
CREATE OR REPLACE FUNCTION check_reorder_point()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.quantity <= (
    SELECT reorder_point FROM products WHERE id = NEW.product_id
  ) THEN
    INSERT INTO reorder_alerts (product_id, location_id, current_quantity)
    VALUES (NEW.product_id, NEW.location_id, NEW.quantity);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER inventory_reorder_check
AFTER UPDATE OF quantity ON inventory
FOR EACH ROW
EXECUTE FUNCTION check_reorder_point();
```

Inventory Operations:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.jdbc.core.JdbcTemplate;
import java.util.*;

/**
 * Retail store inventory management service.
 * Handles stock adjustments, transfers, and reorder alerts.
 */
@Service
public class InventoryService {

    private final JdbcTemplate db;

    @Autowired
    public InventoryService(JdbcTemplate db) {
        this.db = db;
    }
```

```
    /**
     * Adjust stock quantity with full audit trail.
     * Supports in, out, transfer, and adjustment movements.
     */
    @Transactional
    public void adjustStock(Long productId, Long locationId, int quantity,
                            MovementType movementType, String referenceId)
{

        // Update inventory using UPSERT
        String upsertQuery = """
            INSERT INTO inventory (product_id, location_id, quantity)
            VALUES (?, ?, ?)
            ON CONFLICT (product_id, location_id)
            DO UPDATE SET
                quantity = inventory.quantity + EXCLUDED.quantity,
                last_updated = NOW()
            """;

        db.update(upsertQuery, productId, locationId, quantity);

        // Record movement for audit trail
        String movementQuery = """
            INSERT INTO stock_movements
            (product_id, location_id, movement_type, quantity,
reference_id)
            VALUES (?, ?, ?, ?, ?)
            """;

        db.update(movementQuery, productId, locationId,
            movementType.name().toLowerCase(), quantity, referenceId);
    }

    /**
     * Transfer stock between locations atomically.
     */
    @Transactional
    public void transferStock(Long productId, Long fromLocation,
                              Long toLocation, int quantity) {
        // Generate transfer reference ID
        String transferId = "TXF-" +
UUID.randomUUID().toString().substring(0, 8);

        // Verify source has sufficient stock
        Integer currentStock = db.queryForObject(
            "SELECT quantity FROM inventory WHERE product_id = ? AND
location_id = ?",
            Integer.class, productId, fromLocation);

        if (currentStock == null || currentStock < quantity) {
            throw new InsufficientStockException(
                "Insufficient stock at source location. Available: " +
                (currentStock != null ? currentStock : 0));
```

```
        }

        // Deduct from source location
        adjustStock(productId, fromLocation, -quantity,
            MovementType.TRANSFER, transferId);

        // Add to destination location
        adjustStock(productId, toLocation, quantity,
            MovementType.TRANSFER, transferId);
    }

    /**
     * Get items below reorder point, optionally filtered by location.
     */
    public List<LowStockItem> getLowStockItems(Long locationId) {
        String query = """
            SELECT p.id, p.sku, p.name, p.reorder_point,
p.reorder_quantity,
                   i.quantity, i.location_id, l.name as location_name
            FROM products p
            JOIN inventory i ON p.id = i.product_id
            JOIN locations l ON i.location_id = l.id
            WHERE i.quantity <= p.reorder_point
            """;

        if (locationId != null) {
            query += " AND i.location_id = ?";
            return db.query(query, new LowStockItemRowMapper(),
locationId);
        }

        return db.query(query, new LowStockItemRowMapper());
    }

    /**
     * Get stock level for a product at a specific location.
     */
    public Optional<Integer> getStockLevel(Long productId, Long
locationId) {
        String query = """
            SELECT quantity FROM inventory
            WHERE product_id = ? AND location_id = ?
            """;

        try {
            Integer quantity = db.queryForObject(query, Integer.class,
                productId, locationId);
            return Optional.ofNullable(quantity);
        } catch (EmptyResultDataAccessException e) {
            return Optional.empty();
        }
    }

    /**
```

```
     * Get stock movement history for a product.
     */
    public List<StockMovement> getMovementHistory(Long productId,
                                                  Long locationId,
                                                  int limit) {
        String query = """
            SELECT * FROM stock_movements
            WHERE product_id = ? AND location_id = ?
            ORDER BY created_at DESC
            LIMIT ?
            """;

        return db.query(query, new StockMovementRowMapper(),
            productId, locationId, limit);
    }
}

/**
 * Types of inventory movements.
 */
public enum MovementType {
    IN,           // Stock received
    OUT,          // Stock sold/shipped
    TRANSFER,     // Inter-location transfer
    ADJUSTMENT    // Manual correction
}

/**
 * Low stock item with reorder information.
 */
@Data
public class LowStockItem {
    private Long productId;
    private String sku;
    private String name;
    private int currentQuantity;
    private int reorderPoint;
    private int reorderQuantity;
    private Long locationId;
    private String locationName;
}

/**
 * Stock movement audit record.
 */
@Data
public class StockMovement {
    private Long id;
    private Long productId;
    private Long locationId;
    private MovementType movementType;
    private int quantity;
    private String referenceId;
    private Instant createdAt;
```

```
}

/**
 * Exception for insufficient stock scenarios.
 */
public class InsufficientStockException extends RuntimeException {
    public InsufficientStockException(String message) {
        super(message);
    }
}
```

Summary of All 45 Solutions

This document provides comprehensive system design solutions covering:

Core Systems (1-15):

- Streaming platforms, search systems, file storage
- Booking systems with concurrency control
- Distributed scheduling, payment gateways

Concepts & Patterns (16-23):

- Caching strategies, sharding, consistency models
- Rate limiting, top-K algorithms

Specialized Systems (24-45):

- Reconciliation, flight inventory
- Distributed KV stores, message brokers
- Real-time chat, leaderboards
- Multi-tenant platforms

Key Themes:

- Horizontal scalability via partitioning/sharding
- High availability through replication
- Low latency via multi-level caching
- Strong consistency where needed (transactions)
- Eventual consistency where acceptable (performance)
- Observability and monitoring throughout

Each solution includes: ✅ Architecture diagrams ✅ Database schemas ✅ Implementation code ✅ Trade-off analysis ✅ Scalability considerations

---

End of Complete System Design Solutions (1-45)