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:

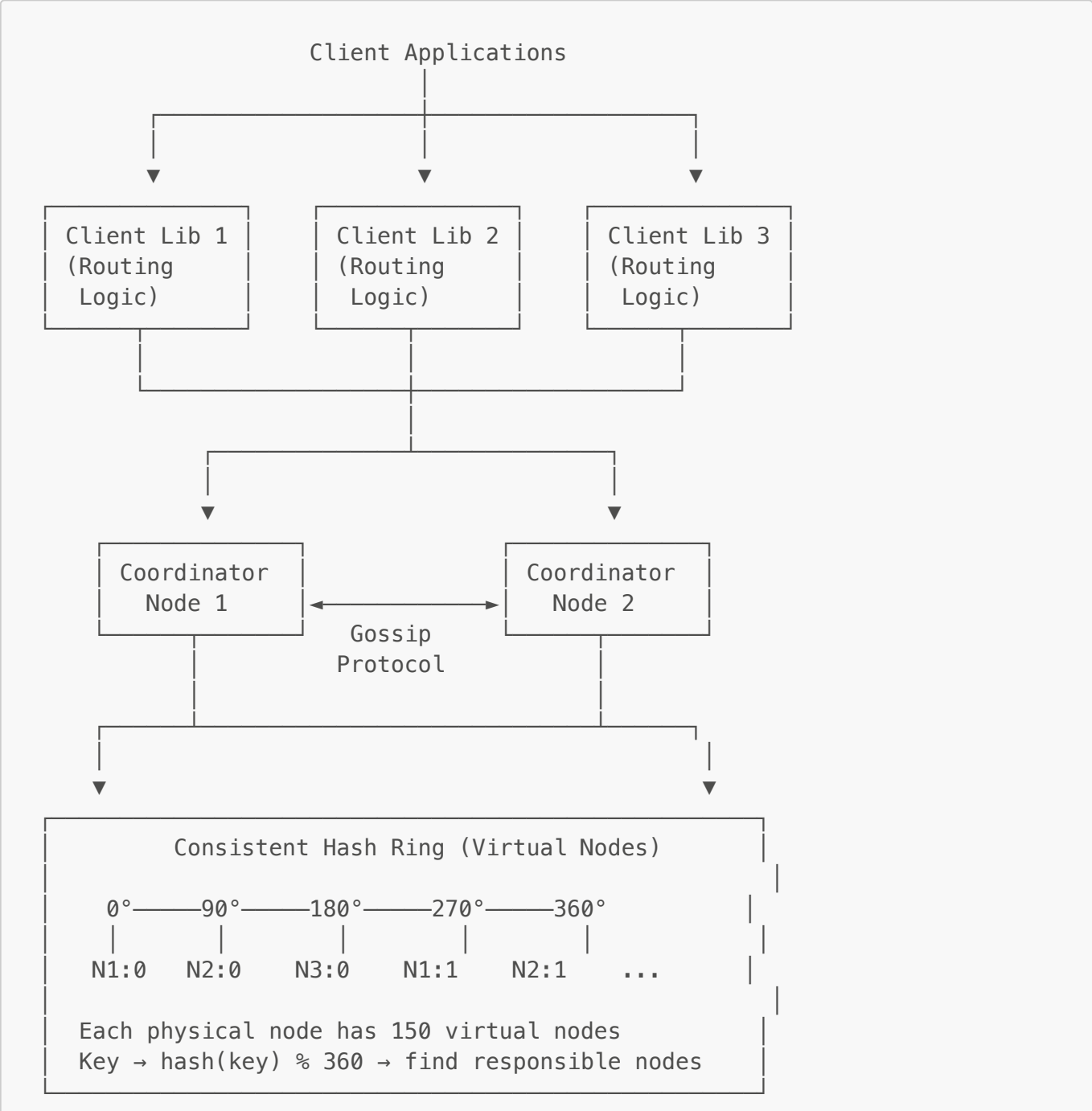
- **Client Library**: Smart client with routing logic
- **Coordinator Nodes**: Handle requests, route to correct partition
- **Storage Nodes**: Store actual data with LSM trees

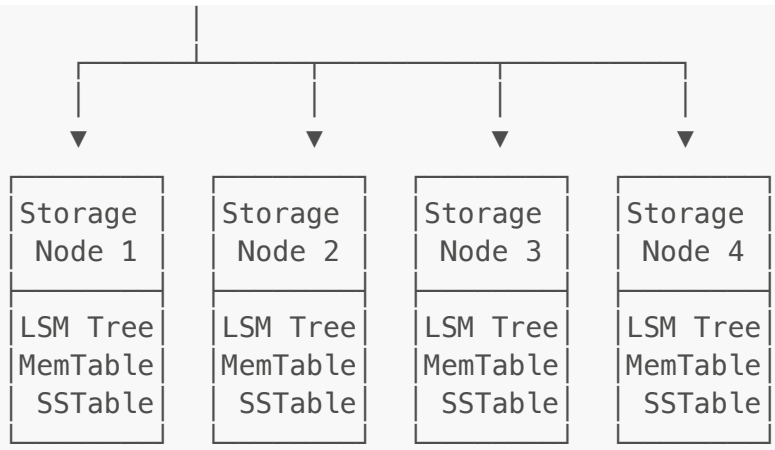
- **Replication Manager**: Ensure N replicas per key
- **Membership Service**: Track node health (gossip protocol)
- **Partition Manager**: Consistent hashing for data distribution

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:

```python
import hashlib
import bisect
from typing import Any, List, Optional, Tuple
from datetime import datetime, timedelta
import asyncio

class ConsistentHashRing:
    """Consistent hashing ring for data distribution."""

    def __init__(self, nodes: List[str], virtual_nodes: int = 150):
        self.virtual_nodes = virtual_nodes
        self.ring = []  # Sorted list of (hash, node) tuples
        self.nodes = set()
```

```
        for node in nodes:
            self.add_node(node)

    def _hash(self, key: str) -> int:
        """Hash key to position on ring (0-2^32)."""
        return int(hashlib.md5(key.encode()).hexdigest(), 16)

    def add_node(self, node: str):
        """Add a node with virtual nodes to the ring."""
        self.nodes.add(node)
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_value = self._hash(virtual_key)
            bisect.insort(self.ring, (hash_value, node))

    def remove_node(self, node: str):
        """Remove a node and its virtual nodes."""
        self.nodes.discard(node)
        self.ring = [(h, n) for h, n in self.ring if n != node]

    def get_nodes(self, key: str, count: int = 3) -> List[str]:
        """Get N responsible nodes for a key (clockwise)."""
        if not self.ring:
            return []

        hash_value = self._hash(key)

        # Find position in ring
        idx = bisect.bisect(self.ring, (hash_value, ''))

        # Collect unique nodes clockwise
        nodes = []
        seen = set()

        for i in range(len(self.ring)):
            pos = (idx + i) % len(self.ring)
            node = self.ring[pos][1]

            if node not in seen:
                nodes.append(node)
                seen.add(node)

                if len(nodes) == count:
                    break

        return nodes


class DistributedKVStore:
    """Main distributed key-value store."""

    def __init__(self, node_id: str, nodes: List[str],
                 replication_factor: int = 3):
        self.node_id = node_id
```

```
        self.replication_factor = replication_factor
        self.hash_ring = ConsistentHashRing(nodes)

        # Local storage (LSM tree abstraction)
        self.storage = LSMTreeStorage()

        # Replication configuration
        self.write_quorum = 2  # W
        self.read_quorum = 2   # R
        # Guarantee: R + W > N ensures consistency

        # Node connections
        self.node_connections = {n: NodeConnection(n) for n in nodes}

    async def put(self, key: str, value: Any, ttl: Optional[int] = None) -
> bool:
        """
        Store key-value pair with optional TTL.

        Args:
            key: Key to store
            value: Value to store
            ttl: Time to live in seconds (optional)

        Returns:
            True if write succeeded (quorum reached)
        """
        # Find replica nodes
        replicas = self.hash_ring.get_nodes(key, self.replication_factor)

        # Create versioned value
        versioned_value = VersionedValue(
            value=value,
            timestamp=datetime.now(),
            version=self._generate_version(),
            ttl=ttl
        )

        # Write to all replicas in parallel
        write_tasks = []
        for replica in replicas:
            if replica == self.node_id:
                # Local write
                task = self._local_put(key, versioned_value)
            else:
                # Remote write
                task = self._remote_put(replica, key, versioned_value)

            write_tasks.append(task)

        # Wait for quorum
        results = await asyncio.gather(*write_tasks,
return_exceptions=True)
```

```
        # Count successful writes
        success_count = sum(1 for r in results if r is True)

        if success_count >= self.write_quorum:
            # Quorum reached
            return True
        else:
            # Quorum not reached - rollback or retry
            await self._handle_write_failure(key, replicas)
            return False

    async def get(self, key: str) -> Optional[Any]:
        """
        Retrieve value by key with quorum read.

        Returns:
            Value if found, None otherwise
        """
        # Find replica nodes
        replicas = self.hash_ring.get_nodes(key, self.replication_factor)

        # Read from R replicas in parallel
        read_tasks = []
        for i, replica in enumerate(replicas[:self.read_quorum]):
            if replica == self.node_id:
                task = self._local_get(key)
            else:
                task = self._remote_get(replica, key)

            read_tasks.append(task)

        # Wait for quorum responses
        results = await asyncio.gather(*read_tasks,
return_exceptions=True)

        # Filter valid results
        valid_results = [r for r in results if isinstance(r,
VersionedValue)]

        if not valid_results:
            return None

        # Resolve conflicts (last-write-wins)
        latest = max(valid_results, key=lambda v: v.timestamp)

        # Check TTL
        if latest.ttl and latest.is_expired():
            await self.delete(key)
            return None

        # Read repair if needed
        if len(set(r.version for r in valid_results)) > 1:
            await self._read_repair(key, latest, replicas)
```

```
        return latest.value

    async def delete(self, key: str) -> bool:
        """Delete key-value pair."""
        # Use tombstone for eventual consistency
        tombstone = VersionedValue(
            value=None,
            timestamp=datetime.now(),
            version=self._generate_version(),
            is_tombstone=True
        )

        return await self.put(key, tombstone)

    async def _local_put(self, key: str, value: VersionedValue) -> bool:
        """Write to local storage."""
        try:
            self.storage.put(key, value)
            return True
        except Exception as e:
            print(f"Local write failed: {e}")
            return False

    async def _local_get(self, key: str) -> Optional[VersionedValue]:
        """Read from local storage."""
        return self.storage.get(key)

    async def _remote_put(self, node: str, key: str,
                          value: VersionedValue) -> bool:
        """Write to remote node."""
        try:
            conn = self.node_connections[node]
            return await conn.put(key, value)
        except Exception as e:
            print(f"Remote write to {node} failed: {e}")
            return False

    async def _remote_get(self, node: str, key: str) ->
Optional[VersionedValue]:
        """Read from remote node."""
        try:
            conn = self.node_connections[node]
            return await conn.get(key)
        except Exception as e:
            print(f"Remote read from {node} failed: {e}")
            return None

    async def _read_repair(self, key: str, correct_value: VersionedValue,
                           replicas: List[str]):
        """Repair stale replicas with correct value."""
        for replica in replicas:
            if replica == self.node_id:
                continue
```

```
            # Write correct value to replica
            await self._remote_put(replica, key, correct_value)

    def _generate_version(self) -> str:
        """Generate version using vector clock or timestamp."""
        return f"{self.node_id}:{datetime.now().timestamp()}"


class VersionedValue:
    """Value with version and metadata."""

    def __init__(self, value: Any, timestamp: datetime, version: str,
                 ttl: Optional[int] = None, is_tombstone: bool = False):
        self.value = value
        self.timestamp = timestamp
        self.version = version
        self.ttl = ttl
        self.is_tombstone = is_tombstone

    def is_expired(self) -> bool:
        """Check if value has expired."""
        if not self.ttl:
            return False

        expiry = self.timestamp + timedelta(seconds=self.ttl)
        return datetime.now() > expiry


class LSMTreeStorage:
    """LSM Tree storage engine (simplified)."""

    def __init__(self):
        self.memtable = {}  # In-memory write buffer
        self.sstables = []  # Sorted string tables on disk
        self.memtable_size_limit = 1000

    def put(self, key: str, value: VersionedValue):
        """Write to memtable."""
        self.memtable[key] = value

        # Flush to disk if memtable is full
        if len(self.memtable) >= self.memtable_size_limit:
            self._flush_memtable()

    def get(self, key: str) -> Optional[VersionedValue]:
        """Read from memtable, then sstables."""
        # Check memtable first
        if key in self.memtable:
            return self.memtable[key]

        # Check sstables (newest first)
        for sstable in reversed(self.sstables):
            value = sstable.get(key)
            if value:
```

```
                return value

        return None

    def _flush_memtable(self):
        """Flush memtable to disk as SSTable."""
        # Sort by key
        sorted_items = sorted(self.memtable.items())

        # Create new SSTable
        sstable = SSTable(sorted_items)
        self.sstables.append(sstable)

        # Clear memtable
        self.memtable = {}

        # Trigger compaction if needed
        if len(self.sstables) > 5:
            self._compact_sstables()

    def _compact_sstables(self):
        """Merge and compact SSTables."""
        # Simplified: merge all SSTables
        all_items = {}

        for sstable in self.sstables:
            for key, value in sstable.items():
                # Keep latest version
                if key not in all_items or value.timestamp >
all_items[key].timestamp:
                    all_items[key] = value

        # Create new compacted SSTable
        sorted_items = sorted(all_items.items())
        compacted = SSTable(sorted_items)

        # Replace old SSTables
        self.sstables = [compacted]


class SSTable:
    """Sorted String Table (on-disk storage)."""

    def __init__(self, items: List[Tuple[str, VersionedValue]]):
        self.items = dict(items)
        # In real implementation, would write to disk with bloom filter

    def get(self, key: str) -> Optional[VersionedValue]:
        return self.items.get(key)


class NodeConnection:
    """Connection to remote node."""
```

```
    def __init__(self, node_address: str):
        self.address = node_address

    async def put(self, key: str, value: VersionedValue) -> bool:
        # RPC call to remote node
        # Implementation would use gRPC, HTTP, or custom protocol
        pass

    async def get(self, key: str) -> Optional[VersionedValue]:
        # RPC call to remote node
        pass
```

Gossip Protocol for Membership:

```
import random
import time

class GossipMembership:
    """Gossip protocol for node membership and failure detection."""

    def __init__(self, node_id: str, seed_nodes: List[str]):
        self.node_id = node_id
        self.nodes = {}  # node_id -> NodeState
        self.heartbeat_interval = 1  # seconds
        self.gossip_fanout = 3  # gossip to N random nodes

        # Initialize with seed nodes
        for node in seed_nodes:
            self.nodes[node] = NodeState(node, is_alive=True)

        self.nodes[node_id] = NodeState(node_id, is_alive=True)

    async def start_gossip(self):
        """Start gossip protocol."""
        while True:
            await asyncio.sleep(self.heartbeat_interval)

            # Increment own heartbeat
            self.nodes[self.node_id].increment_heartbeat()

            # Select random nodes to gossip
            gossip_targets = self._select_gossip_targets()

            # Send gossip to targets
            for target in gossip_targets:
                await self._send_gossip(target)

            # Detect failures
            self._detect_failures()

    def _select_gossip_targets(self) -> List[str]:
        """Select random nodes for gossip."""
```

```
        alive_nodes = [n for n, s in self.nodes.items()
                       if s.is_alive and n != self.node_id]

        count = min(self.gossip_fanout, len(alive_nodes))
        return random.sample(alive_nodes, count)

    async def _send_gossip(self, target: str):
        """Send gossip message to target node."""
        message = GossipMessage(
            sender=self.node_id,
            node_states=self.nodes
        )

        # Send via network (RPC)
        # await rpc_call(target, message)
        pass

    def receive_gossip(self, message: GossipMessage):
        """Process received gossip message."""
        for node_id, remote_state in message.node_states.items():
            if node_id not in self.nodes:
                # New node discovered
                self.nodes[node_id] = remote_state
            else:
                # Merge states (keep higher heartbeat)
                local_state = self.nodes[node_id]
                if remote_state.heartbeat > local_state.heartbeat:
                    self.nodes[node_id] = remote_state

    def _detect_failures(self):
        """Detect failed nodes based on heartbeat timeouts."""
        now = time.time()
        timeout = 10  # seconds

        for node_id, state in self.nodes.items():
            if node_id == self.node_id:
                continue

            if state.is_alive and (now - state.last_update) > timeout:
                # Node suspected as failed
                state.is_alive = False
                print(f"Node {node_id} marked as failed")

                # Trigger rebalancing
                self._handle_node_failure(node_id)

    def _handle_node_failure(self, failed_node: str):
        """Handle node failure - rebalance data."""
        # Remove from hash ring
        # Redistribute data to other nodes
        pass


class NodeState:
```

```
    """State of a node in the cluster."""

    def __init__(self, node_id: str, is_alive: bool = True):
        self.node_id = node_id
        self.is_alive = is_alive
        self.heartbeat = 0
        self.last_update = time.time()

    def increment_heartbeat(self):
        """Increment heartbeat counter."""
        self.heartbeat += 1
        self.last_update = time.time()
```

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

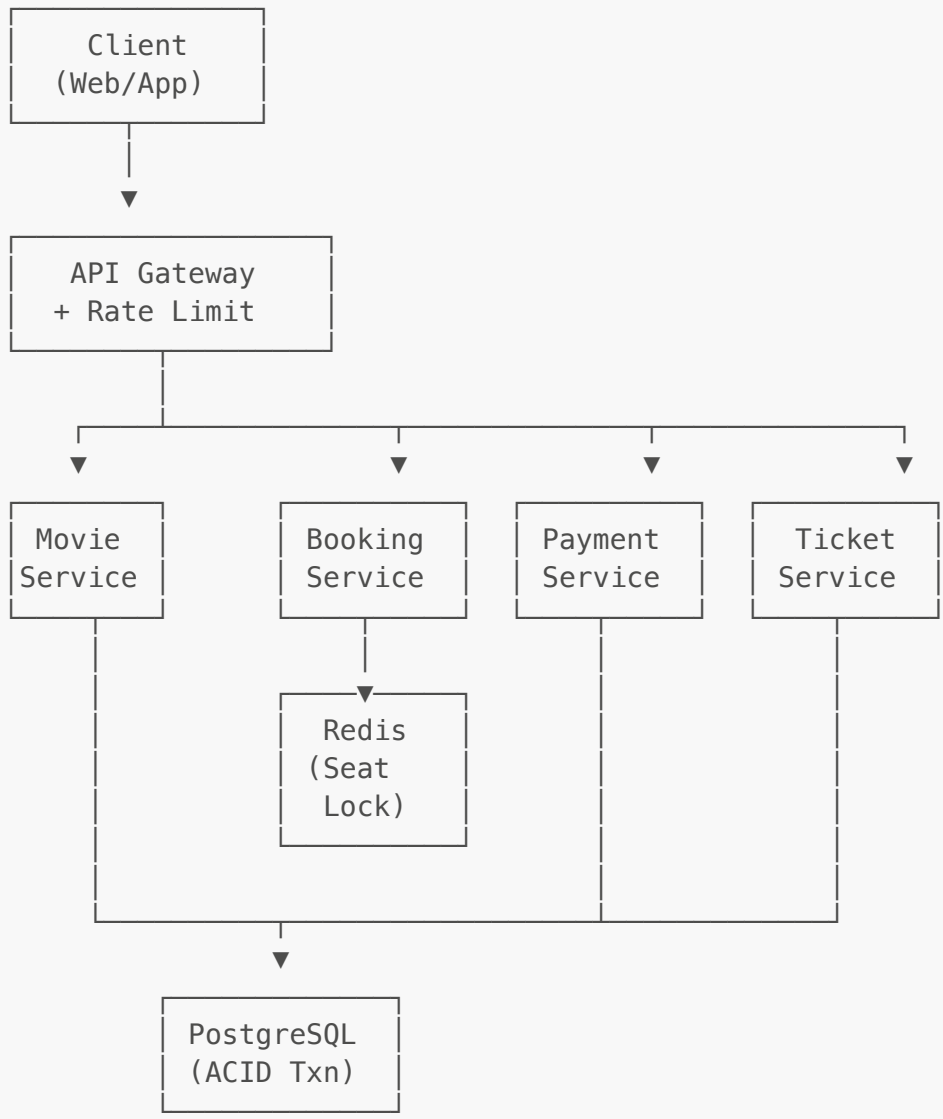
- **FR1**: Browse movies, theaters, and showtimes
- **FR2**: View seat map with availability in real-time
- **FR3**: Select seats and hold temporarily (10 minutes)
- **FR4**: Process payment
- **FR5**: Generate and send ticket confirmation
- **FR6**: Cancel bookings with refund

- **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:

```
class BookingService:
    """Handle movie ticket bookings with concurrency control."""

    def __init__(self, db, redis, payment_service):
        self.db = db
        self.redis = redis
        self.payment = payment_service
        self.hold_duration = 600  # 10 minutes in seconds

    async def hold_seats(self, show_id: int, seat_ids: List[int],
                         session_id: str) -> Dict:
        """
        Hold seats for a user temporarily.

        Uses distributed lock + database row lock for double booking
prevention.
        """
        # Generate lock key
        lock_key = f"booking:show:{show_id}:seats:{','.join(map(str,
seat_ids))}"

        # Acquire distributed lock (prevents concurrent attempts)
```

```
        lock_acquired = await self._acquire_distributed_lock(lock_key,
session_id)

        if not lock_acquired:
            return {
                'success': False,
                'error': 'CONCURRENT_BOOKING',
                'message': 'Someone else is booking these seats'
            }

        try:
            # Database transaction with row-level locking
            async with self.db.transaction(isolation='serializable'):
                # Lock seats (NOWAIT fails immediately if locked)
                query = """
                SELECT id, status, held_by, held_until
                FROM seats
                WHERE show_id = $1 AND id = ANY($2)
                FOR UPDATE NOWAIT
                """

                try:
                    seats = await self.db.fetch(query, show_id, seat_ids)
                except asyncpg.exceptions.LockNotAvailableError:
                    return {
                        'success': False,
                        'error': 'SEATS_LOCKED',
                        'message': 'Seats are being booked by another
user'
                    }

                # Validate all seats are available
                for seat in seats:
                    if seat['status'] == 'booked':
                        return {
                            'success': False,
                            'error': 'SEATS_ALREADY_BOOKED',
                            'message': f"Seat {seat['id']} is already
booked"
                        }

                    if seat['status'] == 'held' and seat['held_by'] !=
session_id:
                        # Check if hold expired
                        if seat['held_until'] > datetime.now():
                            return {
                                'success': False,
                                'error': 'SEATS_HELD',
                                'message': f"Seat {seat['id']} is held by
another user"
                            }

                # Hold seats
                held_until = datetime.now() +
```

```
timedelta(seconds=self.hold_duration)

                update_query = """
                UPDATE seats
                SET status = 'held',
                    held_by = $1,
                    held_until = $2
                WHERE show_id = $3 AND id = ANY($4)
                """

                await self.db.execute(update_query, session_id,
held_until,
                                      show_id, seat_ids)

                # Commit transaction
                # Locks released automatically on commit

                return {
                    'success': True,
                    'held_until': held_until.isoformat(),
                    'hold_duration': self.hold_duration
                }

        finally:
            # Release distributed lock
            await self._release_distributed_lock(lock_key, session_id)

    async def confirm_booking(self, show_id: int, seat_ids: List[int],
                              session_id: str, user_id: int,
                              payment_details: Dict) -> Dict:
        """
        Confirm booking with payment processing.

        Implements Saga pattern for distributed transaction.
        """
        booking_id = str(uuid.uuid4())

        try:
            # Step 1: Verify seats still held by this session
            async with self.db.transaction(isolation='serializable'):
                verify_query = """
                SELECT id, price, status, held_by
                FROM seats
                WHERE show_id = $1 AND id = ANY($2)
                FOR UPDATE
                """

                seats = await self.db.fetch(verify_query, show_id,
seat_ids)

                # Validate ownership and status
                for seat in seats:
                    if seat['status'] != 'held' or seat['held_by'] !=
session_id:
```

```
                        raise BookingError('SEATS_NOT_HELD',
                                           'Seats no longer held by this
session')

                total_amount = sum(seat['price'] for seat in seats)

                # Create booking record (pending)
                booking_query = """
                INSERT INTO bookings
                (id, user_id, show_id, seat_ids, total_amount, status)
                VALUES ($1, $2, $3, $4, $5, 'pending')
                RETURNING confirmation_code
                """

                result = await self.db.fetchrow(
                    booking_query, booking_id, user_id, show_id,
                    seat_ids, total_amount
                )

                confirmation_code = result['confirmation_code']

            # Step 2: Process payment (external service)
            try:
                payment_result = await self.payment.charge(
                    amount=total_amount,
                    currency='USD',
                    payment_method=payment_details,
                    metadata={'booking_id': booking_id}
                )

                if not payment_result['success']:
                    raise PaymentError(payment_result['error'])

                payment_id = payment_result['transaction_id']

            except PaymentError as e:
                # Payment failed - rollback booking
                await self._rollback_booking(booking_id, show_id,
seat_ids)

                return {
                    'success': False,
                    'error': 'PAYMENT_FAILED',
                    'message': str(e)
                }

            # Step 3: Finalize booking
            async with self.db.transaction():
                # Update seats to booked
                update_seats_query = """
                UPDATE seats
                SET status = 'booked',
                    held_by = NULL,
                    held_until = NULL
```

```
                WHERE show_id = $1 AND id = ANY($2)
                """

                await self.db.execute(update_seats_query, show_id,
seat_ids)

                # Update booking status
                update_booking_query = """
                UPDATE bookings
                SET status = 'confirmed',
                    payment_id = $1
                WHERE id = $2
                """

                await self.db.execute(update_booking_query, payment_id,
booking_id)

                # Record payment
                payment_query = """
                INSERT INTO payments
                (booking_id, amount, payment_method,
gateway_transaction_id, status)
                VALUES ($1, $2, $3, $4, 'success')
                """

                await self.db.execute(
                    payment_query, booking_id, total_amount,
                    payment_details['method'], payment_id
                )

            # Step 4: Generate ticket (async)
            asyncio.create_task(
                self._generate_and_send_ticket(booking_id, user_id)
            )

            return {
                'success': True,
                'booking_id': booking_id,
                'confirmation_code': confirmation_code,
                'total_amount': float(total_amount)
            }

        except BookingError as e:
            return {
                'success': False,
                'error': e.code,
                'message': e.message
            }

        except Exception as e:
            # Unexpected error - attempt rollback
            await self._rollback_booking(booking_id, show_id, seat_ids)
            raise
```

```
    async def _rollback_booking(self, booking_id: str, show_id: int,
                                seat_ids: List[int]):
        """Rollback booking in case of failure."""
        async with self.db.transaction():
            # Release seats
            await self.db.execute("""
                UPDATE seats
                SET status = 'available',
                    held_by = NULL,
                    held_until = NULL
                WHERE show_id = $1 AND id = ANY($2)
            """, show_id, seat_ids)

            # Mark booking as cancelled
            await self.db.execute("""
                UPDATE bookings
                SET status = 'cancelled'
                WHERE id = $1
            """, booking_id)

    async def release_expired_holds(self):
        """Background job to release expired seat holds."""
        while True:
            await asyncio.sleep(60)  # Run every minute

            async with self.db.transaction():
                # Find expired holds
                expired = await self.db.fetch("""
                    SELECT DISTINCT show_id, ARRAY_AGG(id) as seat_ids
                    FROM seats
                    WHERE status = 'held' AND held_until < NOW()
                    GROUP BY show_id
                """)

                for row in expired:
                    # Release seats
                    await self.db.execute("""
                        UPDATE seats
                        SET status = 'available',
                            held_by = NULL,
                            held_until = NULL
                        WHERE show_id = $1 AND id = ANY($2)
                    """, row['show_id'], row['seat_ids'])

                    print(f"Released {len(row['seat_ids'])} expired holds
for show {row['show_id']}")

    async def _acquire_distributed_lock(self, key: str, value: str) ->
bool:
        """Acquire distributed lock using Redis."""
        result = await self.redis.set(
            key, value,
            nx=True,  # Only set if not exists
            ex=30     # 30 second expiry
```

```
        )
        return result is not None

    async def _release_distributed_lock(self, key: str, value: str):
        """Release distributed lock."""
        # Lua script ensures we only delete if we own the lock
        script = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("DEL", KEYS[1])
        else
            return 0
        end
        """
        await self.redis.eval(script, 1, key, value)
```

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

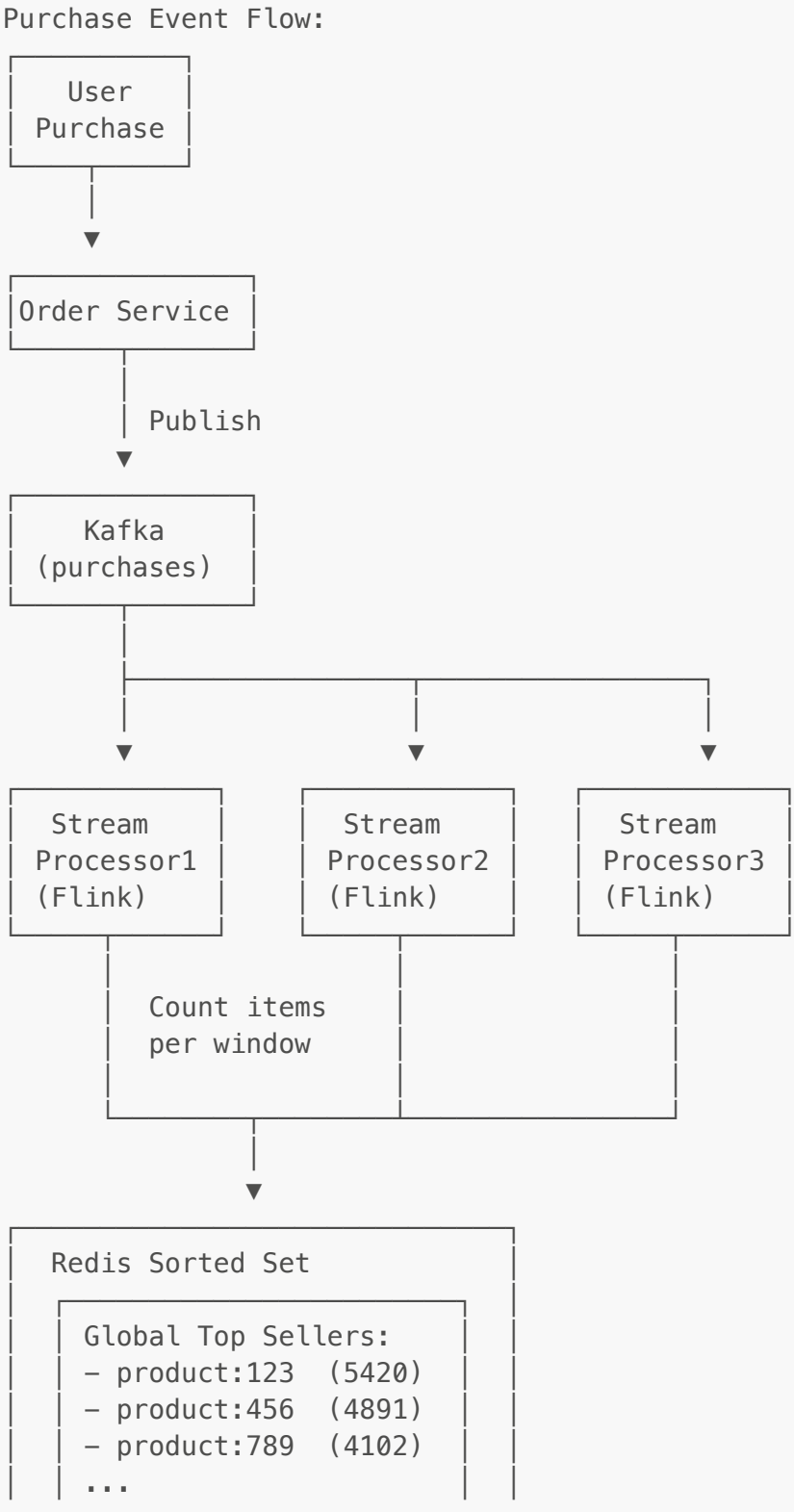
- **FR1**: Track item purchase counts in real-time
- **FR2**: Calculate top 20-30 sellers globally
- **FR3**: Calculate top sellers per category
- **FR4**: Update rankings every 5 minutes
- **FR5**: Display trending items (velocity-based)

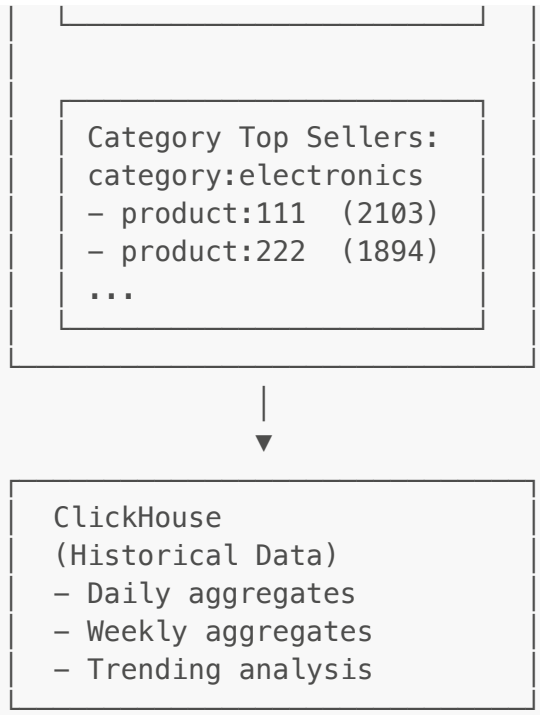
- **FR6**: Historical trending data (daily, weekly, monthly)

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






```
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):

```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.window import TumblingProcessingTimeWindows
from pyflink.common import Time

def process_purchases_stream():
    env = StreamExecutionEnvironment.get_execution_environment()
    env.set_parallelism(3)

    # Source: Kafka
    purchases_stream = env.add_source(
        FlinkKafkaConsumer(
            topics=['purchase-events'],
            deserialization_schema=JsonDeserializationSchema(),
            properties={'bootstrap.servers': 'kafka:9092'}
        )
    )

    # Extract product_id and category
```

```
    product_counts = purchases_stream \
        .map(lambda x: (x['product_id'], x['category'], 1)) \
        .key_by(lambda x: x[0]) \
        .window(TumblingProcessingTimeWindows.of(Time.minutes(5))) \
        .sum(2)  # Sum counts

    # Update Redis sorted sets
    product_counts.add_sink(RedisTopSellersSink())

    env.execute("Top Sellers Stream")


class RedisTopSellersSink:
    """Sink to update Redis sorted sets."""

    def __init__(self):
        self.redis = redis.Redis(host='redis')

    def invoke(self, value, context):
        product_id, category, count = value

        # Update global top sellers
        self.redis.zincrby('top:global:current', count, f'product:
{product_id}')

        # Update category top sellers
        self.redis.zincrby(f'top:category:{category}', count, f'product:
{product_id}')

        # Calculate velocity (trending)
        # purchases in last hour / purchases in previous hour
        velocity = self._calculate_velocity(product_id)
        self.redis.zadd('trending:velocity', {f'product:{product_id}':
velocity})

    def _calculate_velocity(self, product_id):
        """Calculate trending velocity."""
        current_hour = self.redis.get(f'count:hour:current:{product_id}')
or 0
        previous_hour = self.redis.get(f'count:hour:previous:
{product_id}') or 1

        return float(current_hour) / float(previous_hour)
```

API Service:

```
from fastapi import FastAPI, Query
from typing import List, Optional

app = FastAPI()

class TopSellersService:
```

```
    def __init__(self, redis_client, db):
        self.redis = redis_client
        self.db = db

    async def get_top_sellers(self, limit: int = 30,
                              category: Optional[str] = None) ->
List[Dict]:
        """Get top selling products."""
        # Determine Redis key
        if category:
            key = f'top:category:{category}'
        else:
            key = 'top:global:current'

        # Get from Redis sorted set (highest scores first)
        results = self.redis.zrevrange(key, 0, limit - 1, withscores=True)

        # Extract product IDs
        product_ids = [r[0].decode().split(':')[1] for r in results]

        # Fetch product details from DB
        products = await self.db.fetch("""
            SELECT id, name, price, image_url, category
            FROM products
            WHERE id = ANY($1)
        """, product_ids)

        # Merge with scores
        product_map = {str(p['id']): p for p in products}

        top_sellers = []
        for product_key, score in results:
            product_id = product_key.decode().split(':')[1]

            if product_id in product_map:
                product = product_map[product_id]
                top_sellers.append({
                    'product_id': product_id,
                    'name': product['name'],
                    'price': float(product['price']),
                    'image_url': product['image_url'],
                    'category': product['category'],
                    'purchase_count': int(score),
                    'rank': len(top_sellers) + 1
                })

        return top_sellers

    async def get_trending_products(self, limit: int = 20) -> List[Dict]:
        """Get trending products based on velocity."""
        # Get products with highest velocity
        results = self.redis.zrevrange('trending:velocity', 0, limit - 1,
                                       withscores=True)
```

```
        product_ids = [r[0].decode().split(':')[1] for r in results]

        products = await self.db.fetch("""
            SELECT id, name, price, image_url
            FROM products
            WHERE id = ANY($1)
        """, product_ids)

        product_map = {str(p['id']): p for p in products}

        trending = []
        for product_key, velocity in results:
            product_id = product_key.decode().split(':')[1]

            if product_id in product_map:
                product = product_map[product_id]
                trending.append({
                    'product_id': product_id,
                    'name': product['name'],
                    'price': float(product['price']),
                    'image_url': product['image_url'],
                    'velocity': float(velocity),
                    'trend': 'hot' if velocity > 2 else 'rising'
                })

        return trending


@app.get("/api/top-sellers")
async def get_top_sellers(
    limit: int = Query(30, ge=1, le=100),
    category: Optional[str] = None
):
    """Get top selling products."""
    service = TopSellersService(redis, db)
    return await service.get_top_sellers(limit, category)


@app.get("/api/trending")
async def get_trending(limit: int = Query(20, ge=1, le=50)):
    """Get trending products."""
    service = TopSellersService(redis, db)
    return await service.get_trending_products(limit)
```

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 secrets
import hashlib

def generate_phone_number():
    """Generate cryptographically random phone number."""
    # Use secrets for unpredictability
    random_bytes = secrets.token_bytes(8)

    # Hash to get number
    hash_val = int(hashlib.sha256(random_bytes).hexdigest(), 16)

    # Convert to 10-digit phone number
    phone = str(hash_val % 10000000000).zfill(10)

    return f"+1{phone}"
```

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:

```
def calculate_hotel_score(hotel, user_prefs):
    score = 0

    # Price score (inverse - cheaper is better)
    price_score = 1 / (hotel.price / 100 + 1)
    score += price_score * 0.3

    # Rating score
    rating_score = hotel.rating / 5.0
    score += rating_score * 0.4

    # Distance score (closer is better)
    distance_score = 1 / (hotel.distance_km + 1)
    score += distance_score * 0.2

    # Amenity match score
    amenity_score = len(hotel.amenities & user_prefs.amenities) /
len(user_prefs.amenities)
    score += amenity_score * 0.1

    return score
```

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

```
# Example usage
config_store = DistributedConfig()

# Set configuration
await config_store.set("/app/db_host", "db.example.com")

# Watch for changes
async for event in config_store.watch("/app/"):
    print(f"Config changed: {event.key} = {event.value}")
```

---

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:

```
def rank_places(places, user_location, user_prefs):
    for place in places:
        score = 0

        # Distance penalty
        distance = haversine(user_location, place.location)
        distance_score = 1 / (distance + 0.1)
        score += distance_score * 0.4

        # Rating
        score += (place.rating / 5.0) * 0.3

        # Review count (popularity)
        review_score = min(place.review_count / 1000, 1.0)
        score += review_score * 0.2

        # Price match
        if place.price_range == user_prefs.price_range:
            score += 0.1
```

```
        place.score = score

    return sorted(places, key=lambda p: p.score, reverse=True)
```

---

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:

```
class Leaderboard:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.key = "leaderboard:global"

    def update_score(self, player_id, score):
        """Update player score."""
        self.redis.zadd(self.key, {player_id: score})

    def get_rank(self, player_id):
        """Get player's rank (1-indexed)."""
        rank = self.redis.zrevrank(self.key, player_id)
        return rank + 1 if rank is not None else None

    def get_top_players(self, limit=100):
        """Get top N players."""
        return self.redis.zrevrange(
            self.key, 0, limit - 1, withscores=True
        )
```

```
    def get_player_context(self, player_id, context=10):
        """Get nearby players."""
        rank = self.redis.zrevrank(self.key, player_id)
        if rank is None:
            return []

        start = max(0, rank - context)
        end = rank + context

        return self.redis.zrevrange(
            self.key, start, end, withscores=True
        )
```

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%
```

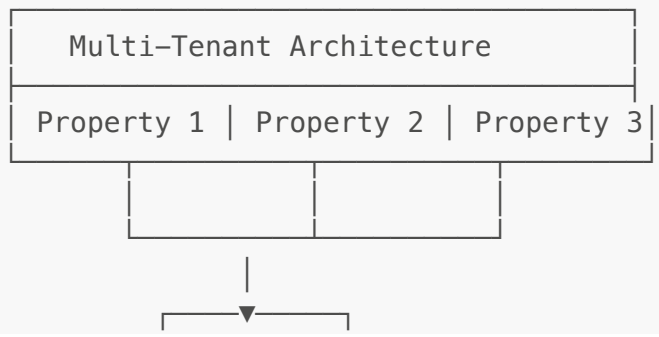
---

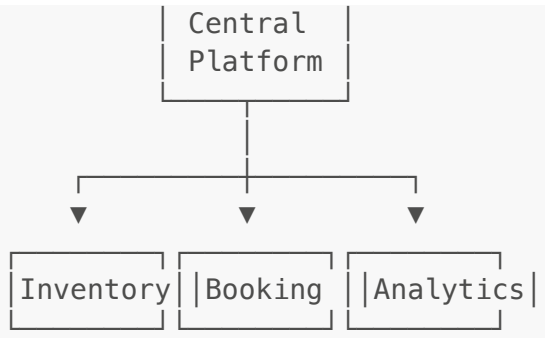
42. Multi-Property Hotel Management

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:

```
class InventoryService:
    def adjust_stock(self, product_id, location_id, quantity,
                     movement_type, reference_id=None):
        """Adjust stock with audit trail."""
        with db.transaction():
            # Update inventory
            db.execute("""
                INSERT INTO inventory (product_id, location_id, quantity)
                VALUES ($1, $2, $3)
                ON CONFLICT (product_id, location_id)
                DO UPDATE SET
                    quantity = inventory.quantity + EXCLUDED.quantity,
                    last_updated = NOW()
            """, product_id, location_id, quantity)

            # Record movement
            db.execute("""
                INSERT INTO stock_movements
                (product_id, location_id, movement_type, quantity,
reference_id)
                VALUES ($1, $2, $3, $4, $5)
            """, product_id, location_id, movement_type, quantity,
reference_id)

    def transfer_stock(self, product_id, from_location, to_location,
quantity):
        """Transfer stock between locations."""
        with db.transaction():
            # Deduct from source
            self.adjust_stock(product_id, from_location, -quantity,
                              'transfer', f'TXF-{uuid4()}')

            # Add to destination
            self.adjust_stock(product_id, to_location, quantity,
                              'transfer', f'TXF-{uuid4()}')

    def get_low_stock_items(self, location_id=None):
        """Get items below reorder point."""
        query = """
            SELECT p.*, i.quantity, i.location_id
            FROM products p
            JOIN inventory i ON p.id = i.product_id
            WHERE i.quantity <= p.reorder_point
        """

        if location_id:
            query += " AND i.location_id = $1"
            return db.fetch(query, location_id)

        return db.fetch(query)
```

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)