Complete System Design Interview Solutions

A comprehensive guide to 45 system design problems with architecture diagrams, trade-offs, and best practices.

Table of Contents

---

1. Music Streaming Application

Problem Overview

Design a music streaming platform that fetches and displays top trending songs with regional filtering, supporting millions of concurrent users with real-time updates and personalized recommendations.

Back-of-the-Envelope Estimation

- **DAU**: 50 million users
- **Peak concurrent users**: 10 million
- **Song requests/sec**: 10M / 86400 × 3 (avg 3 songs/user/day) = ~350 requests/sec (peak: 2000 req/sec)
- **Storage**: 100M songs × 5MB avg = 500TB for audio files
- **Metadata DB**: 100M songs × 10KB metadata = 1TB
- **Bandwidth**: 2000 req/sec × 320kbps = 640 Gbps peak

Functional Requirements

- **FR1**: Users can stream songs with play/pause/skip controls
- **FR2**: Display top trending songs globally and by region
- **FR3**: Search songs by title, artist, album, genre
- **FR4**: Create and manage playlists
- **FR5**: Regional content filtering and recommendations

Non-Functional Requirements

- **Scalability**: Handle 50M DAU with horizontal scaling
- **Availability**: 99.9% uptime (CDN-backed)
- **Latency**: <200ms for song metadata, <2s for audio stream start
- **Consistency**: Eventual consistency for trending data (acceptable delay: 5-15 minutes)

High-Level Architecture

Components:

- **Client**: Web/Mobile apps
- **API Gateway**: Rate limiting, authentication, routing
- **User Service**: Authentication, profiles, preferences
- **Catalog Service**: Song metadata, search indexing

- **Streaming Service**: Audio delivery coordination
- **Trending Service**: Real-time analytics for popular songs
- **Recommendation Service**: ML-based personalized suggestions
- **Databases**: PostgreSQL (metadata), Cassandra (events), Redis (cache)
- **CDN**: Audio file distribution (CloudFront/Akamai)
- **Message Queue**: Kafka for event streaming
- **Object Storage**: S3 for audio files

Data Storage Choices

Data Type	Storage	Justification
Song Metadata	PostgreSQL	Relational data with ACID properties, complex queries
User Listening Events	Cassandra	High write throughput, time-series data
Trending Cache	Redis	Fast read access, TTL support, sorted sets for rankings
Audio Files	S3 + CDN	Blob storage with global distribution
Search Index	Elasticsearch	Full-text search, fuzzy matching

Schema Design:

```
-- PostgreSQL
songs (
  id UUID PRIMARY KEY,
  title VARCHAR(255),
  artist_id UUID,
  album_id UUID,
  duration INT,
  genre VARCHAR(50),
  region VARCHAR(10),
  file_url VARCHAR(500),
  created_at TIMESTAMP
)

artists (
  id UUID PRIMARY KEY,
  name VARCHAR(255),
  bio TEXT,
  country VARCHAR(50)
)

-- Cassandra (events)
listening_events (
  user_id UUID,
  song_id UUID,
  timestamp TIMESTAMP,
  region VARCHAR(10),
  duration_played INT,
  PRIMARY KEY ((region, timestamp), user_id, song_id)
)
```

High-Level Diagram

```
┌─────────────┐
│   Client    │
│ (Web/Mobile)│
└──────┬──────┘
       │
       ▼
┌─────────────────┐
│  API Gateway    │
│ + Auth/Rate Lim │
└────────┬────────┘
         │
    ┌────┴──────────┬──────────────┬──────────────┐
    ▼               ▼              ▼              ▼
┌────────┐   ┌──────────┐   ┌──────────┐   ┌──────────┐
│  User  │   │ Catalog  │   │Streaming │   │ Trending │
│Service │   │ Service  │   │ Service  │   │ Service  │
└───┬────┘   └────┬─────┘   └────┬─────┘   └────┬─────┘
    │             │              │              │
    │        ┌────▼────┐    ┌────▼────┐    ┌────▼────┐
    │        │  Redis  │    │   CDN   │    │  Kafka  │
    │        │  Cache  │    │  + S3   │    │  Stream │
    │        └─────────┘    └─────────┘    └────┬────┘
    │                                           │
    ▼                                           ▼
┌──────────┐                              ┌──────────┐
│PostgreSQL│                              │Cassandra │
│(Metadata)│                              │ (Events) │
└──────────┘                              └──────────┘
```

Trending Calculation Flow:

```
User Listens → Kafka → Streaming Processor
                              ↓
                       Count-Min Sketch
                              ↓
                       Redis Sorted Set (Top 100)
                              ↓
                       Regional Rankings
```

Trade-offs & Assumptions

- **CDN vs Direct Streaming**: CDN adds cost but reduces latency and origin load (95% cache hit rate)
- **Eventual Consistency**: Trending data can be 5-15 min stale; acceptable for better performance

- **Regional Sharding**: Data partitioned by region for compliance and latency; cross-region queries limited
- **Precomputed Rankings**: Rankings updated every 5 minutes; real-time too expensive at scale
- **Assumption**: Most users consume popular content (80/20 rule), making caching highly effective

---

2. Hotel Searching System

Problem Overview

Design a hotel search system that allows users to search hotels by location, dates, price range, and amenities, with support for adding/removing hotels, real-time availability, and high read throughput.

Back-of-the-Envelope Estimation

- **DAU**: 10 million users
- **Hotels in system**: 2 million properties
- **Search requests/sec**: 10M × 5 searches/day / 86400 = ~580 req/sec (peak: 3000 req/sec)
- **Booking writes/sec**: 10M × 0.1 bookings/day / 86400 = ~12 writes/sec
- **Storage**: 2M hotels × 50KB details = 100GB metadata
- **Cache size**: Top 100K hotels × 50KB = 5GB

Functional Requirements

- **FR1**: Search hotels by location (city, coordinates), check-in/out dates
- **FR2**: Filter by price range, star rating, amenities
- **FR3**: Hotel managers can add/update/remove properties
- **FR4**: Real-time availability checking
- **FR5**: Sort results by price, rating, distance

Non-Functional Requirements

- **Scalability**: Support 10M DAU with read-heavy workload
- **Availability**: 99.95% uptime
- **Latency**: <500ms for search results, <100ms for availability check
- **Consistency**: Strong consistency for bookings, eventual for search results

High-Level Architecture

Components:

- **Client**: Web/Mobile
- **API Gateway**: Rate limiting, request routing
- **Search Service**: Query processing, filter application
- **Hotel Service**: CRUD operations for hotel data
- **Inventory Service**: Real-time availability management
- **Geospatial Service**: Location-based filtering
- **Cache**: Redis (multi-layer)
- **Database**: PostgreSQL (main), Elasticsearch (search index)
- **CDN**: Static content (images)

Data Storage Choices

Data Type	Storage	Justification
Hotel Details	PostgreSQL	Relational integrity, complex queries
Search Index	Elasticsearch	Geospatial queries, full-text search, faceted filtering
Availability	Redis + PostgreSQL	Fast read/write, with persistent backup
Images	S3 + CDN	Blob storage with edge caching

Schema:

```
hotels (
  id BIGINT PRIMARY KEY,
  name VARCHAR(255),
  description TEXT,
  address TEXT,
  city VARCHAR(100),
  country VARCHAR(50),
  latitude DECIMAL(10,8),
  longitude DECIMAL(11,8),
  star_rating INT,
  base_price DECIMAL(10,2),
  amenities JSONB,
  created_at TIMESTAMP
)

rooms (
  id BIGINT PRIMARY KEY,
  hotel_id BIGINT REFERENCES hotels(id),
  room_type VARCHAR(50),
  max_occupancy INT,
  price_per_night DECIMAL(10,2),
  total_rooms INT
)

inventory (
  room_id BIGINT,
  date DATE,
  available_rooms INT,
  PRIMARY KEY (room_id, date)
)
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     │
```

```
         ▼
    ┌─────────────┐
    │ API Gateway │
    │ + Rate Limit│
    └──────┬──────┘
           │
    ┌──────┴────────────┬──────────────┐
    ▼                   ▼              ▼
    ┌────────┐     ┌──────────┐   ┌──────────┐
    │ Search │     │  Hotel   │   │Inventory │
    │Service │     │ Service  │   │ Service  │
    └───┬────┘     └────┬─────┘   └────┬─────┘
        │               │              │
        │          ┌────▼───┐     ┌────▼───┐
        │          │ Redis  │     │ Redis  │
        │          │(Hotel) │     │(Avail) │
        │          └────────┘     └────────┘
        │
        ▼
    ┌──────────────┐         ┌──────────┐
    │Elasticsearch │◄────────┤PostgreSQL│
    │ (Geo+Search) │  Sync   │  (Main)  │
    └──────────────┘         └──────────┘

Caching Strategy:
L1: Application cache (recent searches) - 1 min TTL
L2: Redis (popular hotels/cities) - 1 hour TTL
L3: Elasticsearch (all searchable data)
```

Rate Limiting:

- User-based: 100 requests/min
- IP-based: 500 requests/min
- API key-based: 10,000 requests/min (for partners)

Trade-offs & Assumptions

- **Elasticsearch vs PostgreSQL**: Elasticsearch for search speed at cost of storage duplication; PostgreSQL as source of truth
- **Cache Invalidation**: Write-through cache with 1-hour TTL; stale data acceptable for search but not bookings
- **Geospatial Indexing**: PostGIS in PostgreSQL + Elasticsearch geo-queries; redundant but optimized for different use cases
- **Read Replicas**: 5 read replicas for PostgreSQL to handle read load
- **Assumption**: 90% of searches are for top 10K hotels in major cities; aggressive caching effective

---

3. Log/Media Storage System

Problem Overview

Design a unified log and media ingestion system that accepts data from multiple sources (REST APIs, CSV uploads, event streams), processes it, stores efficiently, and provides query capabilities.

Back-of-the-Envelope Estimation

- **Log ingestion rate**: 100K events/sec
- **Media uploads**: 10K files/day (avg 5MB each)
- **Daily log volume**: 100K × 86400 × 1KB = 8.64GB/day → 3.2TB/year
- **Daily media volume**: 10K × 5MB = 50GB/day → 18TB/year
- **Retention**: 90 days hot, 2 years cold
- **Query load**: 1000 queries/sec

Functional Requirements

- **FR1**: Accept logs via REST API, message queues, batch CSV uploads
- **FR2**: Accept media files via multipart upload (images, videos)
- **FR3**: Real-time log processing and aggregation
- **FR4**: Query logs by timestamp, source, level, custom fields
- **FR5**: Provide analytics and alerting on log patterns

Non-Functional Requirements

- **Scalability**: Handle 100K events/sec with burst to 500K
- **Availability**: 99.9% write availability, 99.99% read
- **Latency**: <100ms write acknowledgment, <1s query response
- **Durability**: No data loss (at-least-once delivery)

High-Level Architecture

Components:

- **Ingestion Layer**: API Gateway, File Upload Service, Kafka Connect
- **Processing Layer**: Stream processors (Flink/Spark Streaming)
- **Storage Layer**: Elasticsearch (logs), S3 (media + archive)
- **Query Layer**: Kibana, Custom API
- **Monitoring**: Prometheus + Grafana

Data Storage Choices

Data Type	Storage	Justification
Hot Logs (90 days)	Elasticsearch	Fast search, time-series optimization
Cold Logs (>90 days)	S3 + Athena	Cost-effective archival with query capability
Media Files	S3 + CloudFront	Object storage with CDN for access
Metadata	PostgreSQL	Relational queries for media catalog
Stream Buffer	Kafka	Durable message queue with replay

High-Level Diagram

```
Input Sources:
┌──────────┐   ┌──────────┐   ┌──────────┐
│REST API  │   │CSV Upload│   │  Events  │
└────┬─────┘   └────┬─────┘   └────┬─────┘
     │              │              │
     └──────────────┼──────────────┘
                    │
                    ▼
         ┌──────────────┐
         │ API Gateway  │
         │ + Validation │
         └──────┬───────┘
                │
                ▼
         ┌──────────────┐
         │    Kafka     │
         │  (Buffer)    │
         └───┬──────────┘
             │
         ┌───┴────────┐
         │   Flink    │
         │ Processing │
         └───┬────┬───┘
             │    │
     ┌───────┘    └────────┐
     ▼                     ▼
┌─────────────┐   ┌────────────┐
│Elasticsearch│   │     S3     │
│ (Hot Logs)  │   │ (Media +   │
└─────┬───────┘   │  Archive)  │
      │           └────────────┘
      │
      ▼
┌─────────────┐
│   Kibana    │
│  (Query)    │
└─────────────┘

Lifecycle:
Logs → Hot (Elasticsearch 90d) → Archive (S3 + compress)
Media → S3 (immediate) → Glacier (>1 year)
```

Data Flow:

```
1. API/CSV/Event → Validation → Kafka Topic
2. Kafka → Flink Consumer
3. Flink → Transform + Enrich → Fan-out:
   - Elasticsearch (searchable logs)
   - S3 (raw backup)
   - Metrics aggregator → Prometheus
4. TTL Process: ES (90d) → S3 archive
```

Trade-offs & Assumptions

- **Kafka Buffer**: Adds latency (50-100ms) but provides durability and replay capability
- **Elasticsearch Cost**: Expensive for large volumes; archive to S3 after 90 days
- **Media Processing**: Async processing (thumbnails, transcoding) to avoid blocking uploads
- **Schema Evolution**: Use Avro for logs to handle schema changes gracefully
- **Assumption**: 80% of queries target last 7 days of data; optimize hot storage for this window

---

4. Flight Search System

Problem Overview

Design a flight search system aggregating data from multiple third-party providers with metered APIs, handling dynamic real-time price changes, and optimizing for cost and latency.

Back-of-the-Envelope Estimation

- **DAU**: 5 million users
- **Search requests/sec**: 5M × 3 searches/day / 86400 = ~175 req/sec (peak: 1000 req/sec)
- **Third-party APIs**: 10 providers, each with rate limits (100 req/sec)
- **API cost**: $0.001 per request → $175/sec × 86400 = $15K/day if no caching
- **Cache hit rate target**: 70% → Actual cost: $4.5K/day
- **Response time target**: <2 seconds end-to-end

Functional Requirements

- **FR1**: Search flights by origin, destination, dates, passengers
- **FR2**: Aggregate results from multiple providers
- **FR3**: Display real-time pricing and availability
- **FR4**: Filter by price, duration, stops, airline
- **FR5**: Handle booking redirects to provider sites

Non-Functional Requirements

- **Scalability**: Handle 1000 searches/sec peak load
- **Availability**: 99.9% uptime
- **Latency**: <2s for aggregated results
- **Cost Optimization**: Minimize API calls through intelligent caching
- **Consistency**: Eventual consistency acceptable (prices may be stale by 1-2 minutes)

High-Level Architecture

Components:

- **Client**: Web/Mobile apps

- **API Gateway**: Rate limiting, authentication
- **Search Orchestrator**: Parallel API fan-out, result aggregation
- **Provider Adapters**: Normalize responses from different APIs
- **Cache Layer**: Redis (multi-level)
- **Rate Limiter**: Per-provider request throttling
- **Price Tracker**: Monitor price changes, update cache
- **Database**: PostgreSQL (routes, airports), Redis (cache)

Data Storage Choices

Data Type	Storage	Justification
Popular Routes Cache	Redis	Sub-millisecond access, TTL support
Airport/Airline Data	PostgreSQL	Static reference data, complex queries
Search Results	Redis	Short TTL (2-5 min), high throughput
Provider Metadata	PostgreSQL	Configuration, rate limits, credentials
Analytics	ClickHouse	Time-series queries, cost analysis

Caching Strategy:

```
L1: Recent identical searches (1 min TTL)
L2: Popular routes (5 min TTL)
L3: Airport pairs by day (15 min TTL)

Cache Key: hash(origin, dest, date, passengers, filters)
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌─────────────┐
│API Gateway  │
└──────┬──────┘
       │
       ▼
┌────────────────────┐
│Search Orchestrator │────┐
└─────────┬──────────┘    │  Check Cache
          │               │
      ┌───▼────────┐      │
      │   Redis    │◄─────┘
      │   Cache    │
      └────────────┘
```

```
            │ Cache Miss
            ▼
   ┌─────────────────────┐
   │ Provider Adapters   │
   └──┬─────┬──────┬────┘
      │     │      │
      ▼     ▼      ▼
   ┌────┐┌────┐┌────┐
   │API1││API2││API3│ (Rate Limited)
   └────┘└────┘└────┘

  Rate Limiter (Per Provider):
   ┌──────────────┐
   │ Token Bucket │
   │ 100 req/sec  │
   │ per provider │
   └──────────────┘

  Response Flow:
  APIs → Normalize → Dedupe → Sort → Cache → Client
```

Provider Integration Pattern:

```
async function searchFlights(params) {
  // 1. Check cache
  const cached = await cache.get(cacheKey);
  if (cached && !cached.isStale()) return cached;

  // 2. Fan-out to providers (parallel)
  const providers = ['api1', 'api2', 'api3'];
  const promises = providers.map(p =>
    rateLimiter.execute(p, () => adapter[p].search(params))
  );

  // 3. Race with timeout
  const results = await Promise.allSettled(promises, {timeout: 1500});

  // 4. Aggregate and cache
  const aggregated = normalize(results);
  await cache.set(cacheKey, aggregated, TTL);

  return aggregated;
}
```

Trade-offs & Assumptions

- **Cache Staleness**: 2-5 min stale prices acceptable; fresh prices too expensive
- **Parallel vs Sequential**: Parallel API calls reduce latency but increase provider load
- **Timeout Strategy**: 1.5s timeout per provider to ensure <2s total response
- **Rate Limiting**: Token bucket per provider to stay within limits; queue overflow = skip provider

- **Assumption**: 70% cache hit rate based on popular routes (top 1000 routes = 80% of traffic)
- **Cost vs Freshness**: Longer cache TTL reduces cost but increases booking failures due to stale prices

---

5. YouTube

Problem Overview

Design a video sharing platform where registered users can upload videos and any user can search and view content, supporting billions of videos and millions of concurrent viewers.

Back-of-the-Envelope Estimation

- **DAU**: 500 million users
- **Video uploads**: 500 hours/min = 30K hours/day
- **Video views**: 1 billion views/day
- **Storage**: 30K hours × 60 min × 5GB/hour = 9PB/day raw (before compression)
- **Bandwidth**: 1B views × 10 min avg × 5Mbps = 50 Petabits/day = 580 Gbps average
- **QPS**: 1B views / 86400 = ~12K views/sec (peak: 100K/sec)

Functional Requirements

- **FR1**: Registered users upload videos (multiple formats, up to 12 hours)
- **FR2**: All users can search videos by title, tags, description
- **FR3**: All users can view videos with adaptive bitrate streaming
- **FR4**: Display video metadata, comments, likes/dislikes
- **FR5**: Recommend related videos

Non-Functional Requirements

- **Scalability**: Support 500M DAU, 100K concurrent uploads
- **Availability**: 99.99% uptime for viewing, 99.9% for uploads
- **Latency**: <200ms for metadata, <2s for video start
- **Consistency**: Eventual consistency for views/likes, strong for uploads

High-Level Architecture

Components:

- **Client**: Web, Mobile, Smart TV apps
- **API Gateway**: Authentication, rate limiting
- **Upload Service**: Chunked upload handling, resumable
- **Transcoding Service**: Convert to multiple formats/resolutions
- **Video Service**: Metadata management
- **Streaming Service**: Adaptive bitrate delivery
- **Search Service**: Full-text indexing
- **Recommendation Service**: ML-based suggestions
- **CDN**: Global video distribution
- **Storage**: Object storage (S3/GCS) for videos

- **Databases**: PostgreSQL (metadata), Cassandra (analytics)

Data Storage Choices

Data Type	Storage	Justification
Video Files	S3/GCS + CDN	Blob storage with global edge caching
Metadata	PostgreSQL	ACID for ownership, complex queries
Views/Likes/Comments	Cassandra	High write throughput, eventual consistency OK
Search Index	Elasticsearch	Full-text search, ranking
User Sessions	Redis	Fast state management
Thumbnails	S3 + CDN	Image CDN optimization

Schema:

```
-- PostgreSQL
videos (
  id UUID PRIMARY KEY,
  user_id UUID,
  title VARCHAR(255),
  description TEXT,
  duration INT,
  upload_date TIMESTAMP,
  status VARCHAR(20), -- processing, ready, failed
  privacy VARCHAR(20) -- public, unlisted, private
)

-- Cassandra
video_views (
  video_id UUID,
  timestamp TIMESTAMP,
  user_id UUID,
  watch_duration INT,
  PRIMARY KEY ((video_id), timestamp, user_id)
)
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌─────────────┐
│ API Gateway │
└──────┬──────┘
```

```
          │
    ┌─────┴──────────┬───────────────┬───────────────┐
    ▼                ▼               ▼               ▼
┌──────────┐  ┌──────────┐  ┌───────────┐  ┌───────────┐
│  Upload  │  │  Video   │  │ Streaming │  │  Search   │
│ Service  │  │ Service  │  │  Service  │  │  Service  │
└────┬─────┘  └────┬─────┘  └─────┬─────┘  └─────┬─────┘
     │             │              │              │
     ▼             ▼              │              ▼
┌──────────┐  ┌──────────┐        │       ┌───────────────┐
│   S3     │  │  Redis   │        │       │ Elasticsearch │
│  (Raw)   │  │  (Meta)  │        │       └───────────────┘
└────┬─────┘  └──────────┘        │
     │                            │
     ▼                            │
┌──────────────┐                  │
│ Transcoding  │                  │
│   Queue      │                  │
└──────┬───────┘                  │
       │                          │
       ▼                          ▼
┌──────────────┐         ┌──────────────┐
│  Transcoder  │         │ CDN (Video)  │
│   Workers    │         │ Multi-region │
└──────┬───────┘         └──────────────┘
       │
       ▼
┌──────────────┐
│     S3       │
│(Transcoded)  │
└──────────────┘

Upload Flow:
1. Client → Upload Service (chunked)
2. Upload Service → S3 (raw)
3. S3 Event → SQS → Transcoding Workers
4. Workers → Transcode (1080p, 720p, 480p, 360p)
5. Workers → S3 (transcoded) → CDN Invalidation
6. Update video status: processing → ready

View Flow:
1. Client → Streaming Service
2. Streaming Service → CDN
3. CDN → Adaptive bitrate (HLS/DASH)
4. Log view event → Cassandra (async)
```

Transcoding Pipeline:

```
Raw Upload (1080p@30fps)
    ↓
┌────────────────────┐
│ Extract audio      │
│ Generate formats: │
│ - 1080p (5Mbps)   │
│ - 720p (2.5Mbps)  │
│ - 480p (1Mbps)    │
│ - 360p (500Kbps)  │
└─────────┬─────────┘
          │
          ↓
┌───────────────────┐
│ Create thumbnails │
│ Generate preview  │
└─────────┬─────────┘
          │
          ↓
    CDN Upload
```

Trade-offs & Assumptions

- **Transcoding Delay**: Videos available after 5-30 min depending on length; acceptable for UGC platform
- **CDN Cost**: 90% of bandwidth cost but necessary for global low-latency delivery
- **Storage Redundancy**: 3x replication for durability; deleted videos soft-deleted (30 day retention)
- **View Counting**: Eventual consistency (5-10 min delay) acceptable; prevents spam with rate limiting
- **Recommendation**: Collaborative filtering + content-based; updated daily (not real-time)
- **Assumption**: 80% of views are for 10% of videos (power law); aggressive caching effective

---

6. Hotel Booking with Proximity Search

Problem Overview

Design a hotel booking system with emphasis on proximity-based search, allowing users to find hotels near specific locations (coordinates, landmarks) efficiently at scale.

Back-of-the-Envelope Estimation

- **Hotels**: 2 million properties worldwide
- **DAU**: 8 million users
- **Search requests/sec**: 8M × 4 searches/day / 86400 = ~370 req/sec (peak: 2000 req/sec)
- **Proximity queries**: 90% of searches use location-based filtering
- **Radius**: Most searches within 5-50km radius
- **Bookings/day**: 8M × 0.05 = 400K bookings

Functional Requirements

- **FR1**: Search hotels by coordinates with radius (e.g., within 10km)
- **FR2**: Search by landmarks (e.g., "near Eiffel Tower")
- **FR3**: Real-time availability and pricing
- **FR4**: Book rooms with payment processing
- **FR5**: Sort by distance, price, rating

Non-Functional Requirements

- **Scalability**: Handle 2000 proximity searches/sec
- **Availability**: 99.95% uptime
- **Latency**: <300ms for proximity search results
- **Accuracy**: Distance calculation within 1% error
- **Consistency**: Strong consistency for bookings, eventual for search

High-Level Architecture

Components:

- **Client**: Web/Mobile
- **API Gateway**: Rate limiting, routing
- **Geospatial Service**: Proximity calculations, indexing
- **Hotel Service**: CRUD operations
- **Booking Service**: Reservation management
- **Payment Service**: Transaction processing
- **Database**: PostgreSQL + PostGIS, Redis
- **Search Index**: Elasticsearch with geo-queries

Data Storage Choices

Data Type	Storage	Justification
Hotel Locations	PostgreSQL + PostGIS	Geospatial indexing (R-tree), complex queries
Search Cache	Redis + GeoHash	Fast proximity lookups, TTL support
Hotel Details	PostgreSQL	Relational data, ACID properties
Bookings	PostgreSQL	Strong consistency required
Search Index	Elasticsearch	Geo-queries with filters

Geospatial Indexing Strategies:

1. **PostGIS (PostgreSQL)**: R-tree index for precise distance queries
2. **Geohash (Redis)**: Approximate proximity with prefix matching
3. **Quadtree/S2**: Hierarchical spatial indexing

Schema:

```
-- PostgreSQL with PostGIS
hotels (
  id BIGINT PRIMARY KEY,
  name VARCHAR(255),
  description TEXT,
  address TEXT,
  location GEOGRAPHY(POINT, 4326), -- PostGIS type
  star_rating INT,
  base_price DECIMAL(10,2),
  amenities JSONB
)
```

```
-- GiST index for geospatial queries
CREATE INDEX idx_hotel_location ON hotels USING GIST(location);

-- Proximity query
SELECT id, name,
       ST_Distance(location, ST_MakePoint(lon, lat)::geography) AS
distance
FROM hotels
WHERE ST_DWithin(
  location,
  ST_MakePoint(lon, lat)::geography,
  10000  -- 10km in meters
)
ORDER BY distance
LIMIT 50;
```

Geohash Caching:

```
/**
 * Cache hotels by geohash prefix for efficient proximity lookups.
 * Uses geohash encoding to create spatial index keys for Redis caching.
 */
public class GeohashHotelCache {

    private final RedisTemplate<String, List<Hotel>> redisTemplate;
    private final HotelRepository hotelRepository;
    private static final int GEOHASH_PRECISION = 6; // ~1.2km cell
    private static final long CACHE_TTL_SECONDS = 3600; // 1 hour

    public GeohashHotelCache(RedisTemplate<String, List<Hotel>>
redisTemplate,
                            HotelRepository hotelRepository) {
        this.redisTemplate = redisTemplate;
        this.hotelRepository = hotelRepository;
    }

    public List<Hotel> getHotelsByGeohash(double lat, double lon, double
radiusKm) {
        // Encode coordinates to geohash with specified precision
        String geohash = GeoHash.encodeHash(lat, lon, GEOHASH_PRECISION);

        // Get adjacent cells for complete coverage at boundaries
        List<String> neighbors = GeoHash.getNeighbors(geohash);
        neighbors.add(geohash);

        String cacheKey = String.format("hotels:geo:%s", geohash);

        // Check cache first
        List<Hotel> cached = redisTemplate.opsForValue().get(cacheKey);
        if (cached != null) {
            return filterByDistance(cached, lat, lon, radiusKm);
        }

        // Cache miss - query database and populate cache
        List<Hotel> hotels = hotelRepository.findByGeohash(geohash);
        redisTemplate.opsForValue().set(cacheKey, hotels,
            Duration.ofSeconds(CACHE_TTL_SECONDS));

        return filterByDistance(hotels, lat, lon, radiusKm);
    }

    private List<Hotel> filterByDistance(List<Hotel> hotels,
                                          double lat, double lon,
                                          double radiusKm) {
        return hotels.stream()
            .filter(hotel -> calculateHaversineDistance(
                lat, lon, hotel.getLatitude(), hotel.getLongitude()) <=
radiusKm)
            .collect(Collectors.toList());
    }

    private double calculateHaversineDistance(double lat1, double lon1,
                                               double lat2, double lon2) {
        final double R = 6371.0; // Earth's radius in kilometers
        double dLat = Math.toRadians(lat2 - lat1);
        double dLon = Math.toRadians(lon2 - lon1);
        double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
                   Math.cos(Math.toRadians(lat1)) *
Math.cos(Math.toRadians(lat2)) *
                   Math.sin(dLon / 2) * Math.sin(dLon / 2);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
        return R * c;
    }
}
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌─────────────┐
│ API Gateway │
└──────┬──────┘
       │
   ┌───┴──────────────┬──────────────┐
   ▼                  ▼              ▼
┌────────────┐ ┌─────────┐ ┌──────────┐
│ Geospatial │ │  Hotel  │ │ Booking  │
│  Service   │ │ Service │ │ Service  │
└─────┬──────┘ └────┬────┘ └────┬─────┘
```

```
         │            │             │
         ▼            ▼             ▼
   ┌──────────┐  ┌────────────────────┐
   │  Redis   │  │     PostgreSQL     │
   │ Geohash  │  │    + PostGIS       │
   │  Cache   │  │ (Geo R-tree index) │
   └──────────┘  └────────────────────┘

  Proximity Search Flow:
  1. User: "Hotels near (lat, lon) within 10km"
  2. Generate geohash (precision 6)
  3. Check Redis for geohash + neighbors
  4. If cache miss:
     - PostGIS query with ST_DWithin
     - Cache results by geohash
  5. Filter by distance in-memory
  6. Apply additional filters (price, rating)
  7. Return sorted results

  Geohash Grid (Example):
   ┌────────┬────────┬────────┐
   │  u09   │  u0d   │  u0e   │  Precision 3
   │        │        │        │  (~156km)
   ├────────┼────────┼────────┤
   │  u03   │ *u0b*  │  u0c   │
   │        │        │        │  *Central cell
   ├────────┼────────┼────────┤   + 8 neighbors
   │  u02   │  u08   │  u09   │
   │        │        │        │
   └────────┴────────┴────────┘
```

Distance Calculation:

```
Haversine Formula:
a = sin²(Δlat/2) + cos(lat1) × cos(lat2) × sin²(Δlon/2)
c = 2 × atan2(√a, √(1−a))
distance = R × c  (R = Earth radius = 6371 km)
```

Trade-offs & Assumptions

- **PostGIS vs Geohash**: PostGIS for accuracy, Geohash for cache speed; use both
- **Cache Granularity**: Precision 6 geohash (~1.2km cells) balances cache hit rate and freshness
- **Distance Calculation**: Haversine for <1000km, Vincenty for higher accuracy but slower
- **Neighbor Cells**: Query 9 cells (center + 8 neighbors) to cover edge cases
- **Assumption**: 70% of searches are for urban areas with high hotel density; geohash caching very effective
- **Index Overhead**: PostGIS R-tree index adds 20-30% storage but 100x faster queries

---

7. Distributed Scheduler from RDBMS

Problem Overview

Given an RDBMS table with 500 million records containing URLs and their fetch frequencies, design a distributed scheduler that processes URLs based on their frequency across multiple worker nodes.

Back-of-the-Envelope Estimation

- **Total URLs**: 500 million
- **Frequency distribution**:
  - High (hourly): 10M URLs (2%)
  - Medium (daily): 50M URLs (10%)
  - Low (weekly): 440M URLs (88%)
- **Peak load**: 10M hourly + 50M/24 daily + 440M/168 weekly = ~12K URLs/sec
- **Worker nodes**: 100 nodes → ~120 URLs/node/sec
- **DB size**: 500M × 500 bytes = 250GB

Functional Requirements

- **FR1**: Fetch URLs from table based on frequency (hourly, daily, weekly)
- **FR2**: Distribute work evenly across worker nodes
- **FR3**: Handle worker failures and rebalancing
- **FR4**: Ensure no duplicate processing
- **FR5**: Support dynamic frequency updates

Non-Functional Requirements

- **Scalability**: Handle 500M URLs, scale to 1000 workers
- **Availability**: 99.9% uptime, failover <30 seconds
- **Latency**: Schedule within 1 minute of due time
- **Consistency**: Exactly-once processing per frequency window
- **Fault Tolerance**: Automatic recovery from node failures

High-Level Architecture

Components:

- **Scheduler Master**: Coordination, work distribution
- **Worker Nodes**: URL processing
- **Database**: PostgreSQL (URL table)
- **Message Queue**: Kafka/RabbitMQ (work distribution)
- **Coordination**: ZooKeeper/etcd (leader election, membership)
- **Monitoring**: Metrics collection, alerting

Data Storage Choices

Data Type	Storage	Justification
URL Records	PostgreSQL (partitioned)	Source of truth, complex queries
Work Queue	Kafka	Durable queue, replay capability
Worker State	Redis	Fast state tracking, heartbeats

Data Type	Storage	Justification
Execution Log	Cassandra	High write throughput, audit trail
Coordination	ZooKeeper	Leader election, distributed locks

Schema:

```
-- PostgreSQL (partitioned by frequency)
url_schedule (
  id BIGINT PRIMARY KEY,
  url VARCHAR(2048),
  frequency VARCHAR(20), -- hourly, daily, weekly
  last_processed TIMESTAMP,
  next_run TIMESTAMP,
  priority INT,
  status VARCHAR(20), -- pending, processing, completed, failed
  partition_key INT -- for consistent hashing
)

-- Partitions
CREATE TABLE url_schedule_hourly PARTITION OF url_schedule FOR VALUES IN
('hourly');
CREATE TABLE url_schedule_daily PARTITION OF url_schedule FOR VALUES IN
('daily');
CREATE TABLE url_schedule_weekly PARTITION OF url_schedule FOR VALUES IN
('weekly');

-- Index for scheduler
CREATE INDEX idx_next_run ON url_schedule (next_run, status) WHERE status
= 'pending';
```

High-Level Diagram

```
┌───────────────────┐
│  Scheduler Master │  (Leader elected via ZooKeeper)
│  + Partitioner    │
└─────────┬─────────┘
          │
          ▼
┌─────────────────────┐
│    PostgreSQL       │
│  (500M URL records) │
│   Partitioned by    │
│     frequency       │
└──────────┬──────────┘
           │  Poll & Partition
           │
           ▼
┌───────────────────────┐
```

```
│      Kafka          │
│  Topics per freq:   │
│  - hourly-tasks     │
│  - daily-tasks      │
│  - weekly-tasks     │
└──────────────────────┘
        │
        │
        │ Consume
        │
        │
    ┌───┴──────────────────────────┐
    │                              │
    ▼                              ▼
┌─────────┐    ...          ┌─────────┐
│ Worker1 │                 │Worker100│
│ (Fleet) │                 │ (Fleet) │
└────┬────┘                 └────┬────┘
     │                           │
     └─────────────┬─────────────┘
                   ▼
          ┌─────────────┐
          │  Execution  │
          │     Log     │
          │ (Cassandra) │
          └─────────────┘

Scheduler Flow:
1. Master polls DB: SELECT * FROM url_schedule
   WHERE next_run <= NOW() AND status = 'pending'
   LIMIT 10000
2. Partition by hash(url) % num_workers
3. Publish to Kafka topic by frequency
4. Workers consume, process, acknowledge
5. Update status and next_run in DB

Partitioning Strategy:
hash(url) → Worker ID (consistent hashing)
Ensures same URL always goes to same worker (caching benefit)
```

Worker Assignment:

```java
import java.security.MessageDigest;
import java.util.*;
import java.util.concurrent.*;

/**
 * Consistent Hashing implementation for distributed worker assignment.
 * Uses virtual nodes to ensure even distribution of workload.
 */
public class ConsistentHashRing {

    private final TreeMap<String, String> ring = new TreeMap<>();
    private final List<String> nodes;
    private final int virtualNodes;

    public ConsistentHashRing(List<String> nodes, int virtualNodes) {
        this.nodes = new ArrayList<>(nodes);
        this.virtualNodes = virtualNodes;
        buildRing();
    }

    private void buildRing() {
        for (String node : nodes) {
            for (int i = 0; i < virtualNodes; i++) {
                String key = computeMD5Hash(node + ":" + i);
                ring.put(key, node);
            }
        }
    }

    public String getNodeForKey(String url) {
        if (ring.isEmpty()) {
            return null;
        }

        String hash = computeMD5Hash(url);
        Map.Entry<String, String> entry = ring.ceilingEntry(hash);

        // Wrap around to first node if necessary
        if (entry == null) {
            entry = ring.firstEntry();
        }
        return entry.getValue();
    }

    private String computeMD5Hash(String input) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] digest = md.digest(input.getBytes());
            StringBuilder sb = new StringBuilder();
            for (byte b : digest) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (Exception e) {
            throw new RuntimeException("MD5 hash computation failed", e);
        }
    }
}

/**
 * Distributed URL Scheduler that polls database for due URLs
 * and distributes work across worker nodes using consistent hashing.
 */
public class DistributedScheduler implements Runnable {

    private final DataSource dataSource;
    private final KafkaProducer<String, URLRecord> kafkaProducer;
    private final ConsistentHashRing hashRing;
    private final int batchSize;
    private final long pollIntervalMs;
    private volatile boolean running = true;

    public DistributedScheduler(DataSource dataSource,
                                KafkaProducer<String, URLRecord>
kafkaProducer,
                                List<String> workerNodes,
                                int batchSize,
                                long pollIntervalMs) {
        this.dataSource = dataSource;
        this.kafkaProducer = kafkaProducer;
        this.hashRing = new ConsistentHashRing(workerNodes, 150);
        this.batchSize = batchSize;
        this.pollIntervalMs = pollIntervalMs;
    }

    @Override
    public void run() {
        while (running) {
            try {
                scheduleUrls();
                Thread.sleep(pollIntervalMs);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            } catch (Exception e) {
                // Log error and continue
                System.err.println("Scheduling error: " + e.getMessage());
            }
        }
    }

    private void scheduleUrls() throws SQLException {
        String query = """
            SELECT id, url, frequency
            FROM url_schedule
            WHERE next_run <= NOW()
              AND status = 'pending'
            ORDER BY next_run, priority
            LIMIT ?
            """;

        try (Connection conn = dataSource.getConnection();
             PreparedStatement stmt = conn.prepareStatement(query)) {

            stmt.setInt(1, batchSize);
            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {
                URLRecord record = new URLRecord(
                    rs.getLong("id"),
                    rs.getString("url"),
                    rs.getString("frequency")
                );

                // Determine target worker using consistent hashing
                String worker = hashRing.getNodeForKey(record.getUrl());
                String topic = record.getFrequency() + "-tasks";

                // Publish to Kafka with worker as partition key
                ProducerRecord<String, URLRecord> kafkaRecord =
                    new ProducerRecord<>(topic, worker, record);
                kafkaProducer.send(kafkaRecord);

                // Update status to processing
                updateStatusToProcessing(conn, record.getId());
            }
        }
    }

    private void updateStatusToProcessing(Connection conn, long id)
            throws SQLException {
        String updateQuery = "UPDATE url_schedule SET status =
'processing' WHERE id = ?";
        try (PreparedStatement stmt = conn.prepareStatement(updateQuery))
{
            stmt.setLong(1, id);
            stmt.executeUpdate();
        }
    }

    public void shutdown() {
        running = false;
    }
}

/**
 * Data class representing a URL record for scheduling.
 */
public record URLRecord(long id, String url, String frequency) {}
```

Failure Handling:

```
Worker Failure Detection:
- Heartbeat every 5 seconds to Redis
- Master checks heartbeats every 10 seconds
- If no heartbeat for 30 seconds → mark worker as dead
- Rebalance: redistribute URLs from dead worker
- Kafka consumer group rebalancing handles message reassignment

Message Timeout:
- Worker claims message with visibility timeout (5 min)
- If not ack'd within timeout → message redelivered
- Prevents stuck messages

Duplicate Prevention:
- DB status field ensures only one worker processes URL
- Optimistic locking: UPDATE WHERE status = 'pending'
- If UPDATE affects 0 rows → already claimed by another worker
```

Trade-offs & Assumptions

- **Polling vs Push**: Polling DB adds latency (10s) but simpler than change data capture
- **Partition Count**: 100 partitions (= workers) limits scalability but simplifies routing
- **Kafka vs Direct**: Kafka adds complexity but provides durability and replay
- **Consistent Hashing**: Same URL → same worker enables caching but creates hotspots
- **Assumption**: Frequency distribution is stable (90% low-frequency); optimize for batch processing
- **DB Load**: 10K queries every 10 seconds = 1K QPS; add read replicas if needed

---

8. Payment Gateway System

Problem Overview

Design a payment gateway for processing transactions with high scalability, exactly-once Kafka message processing, and integration with multiple payment providers (cards, wallets, UPI).

Back-of-the-Envelope Estimation

- **Transactions/day**: 10 million
- **Peak TPS**: 10M / 86400 × 5 (peak factor) = ~580 TPS
- **Average transaction value**: $50
- **Daily transaction volume**: $500 million
- **Success rate**: 85% (15% failures/retries)
- **Message throughput**: 580 TPS × 2 (request + response) = 1160 msg/sec

Functional Requirements

- **FR1**: Process payments (credit/debit cards, wallets, UPI)
- **FR2**: Support refunds and chargebacks
- **FR3**: Exactly-once transaction processing
- **FR4**: Real-time transaction status updates
- **FR5**: Webhook notifications to merchants

Non-Functional Requirements

- **Scalability**: Handle 10M transactions/day, scale to 100M
- **Availability**: 99.99% uptime (4.38 min downtime/month)
- **Latency**: <2 seconds for transaction response
- **Consistency**: Exactly-once processing, no double charges
- **Durability**: Zero transaction data loss
- **Security**: PCI DSS compliance

High-Level Architecture

Components:

- **Client**: Merchant apps/websites
- **API Gateway**: TLS termination, rate limiting
- **Payment Service**: Transaction orchestration
- **Provider Adapters**: Integration with payment networks
- **Transaction DB**: PostgreSQL (ACID transactions)
- **Message Queue**: Kafka (exactly-once semantics)
- **Idempotency Service**: Deduplication
- **Webhook Service**: Merchant notifications
- **Fraud Detection**: Real-time risk scoring
- **Reconciliation**: Daily settlement matching

Data Storage Choices

Data Type	Storage	Justification
Transactions	PostgreSQL	ACID properties, strong consistency
Idempotency Keys	Redis	Fast lookups, TTL for cleanup
Event Log	Kafka	Durable event streaming, exactly-once
Audit Trail	Cassandra	High write throughput, immutable log
Session State	Redis	Fast token validation
Analytics	ClickHouse	OLAP queries, reporting

Schema:

```
-- PostgreSQL
transactions (
  id UUID PRIMARY KEY,
  idempotency_key VARCHAR(64) UNIQUE,
  merchant_id UUID,
  amount DECIMAL(15,2),
  currency VARCHAR(3),
  status VARCHAR(20), -- pending, processing, success, failed
  payment_method VARCHAR(50),
  provider VARCHAR(50),
  provider_transaction_id VARCHAR(100),
  created_at TIMESTAMP,
  updated_at TIMESTAMP,
  metadata JSONB
)

-- Index for idempotency
CREATE UNIQUE INDEX idx_idempotency ON transactions(idempotency_key);
```

```sql
-- State transitions
CREATE TYPE txn_status AS ENUM ('pending', 'processing', 'authorized',
                                'captured', 'failed', 'refunded');
```

High-Level Diagram

```
┌──────────────┐
│   Merchant   │
│   Frontend   │
└──────┬───────┘
       │
       ▼
┌──────────────────┐
│   API Gateway    │
│ + Rate Limiting  │
│ + TLS/Auth       │
└────────┬─────────┘
         │
         ▼
┌────────────────────┐
│  Payment Service   │
│  + Idempotency     │
└─────────┬──────────┘
          │
     ┌────┴──────────────┐
     │                   │
     ▼                   ▼
┌──────────┐        ┌──────────┐
│  Redis   │        │  Kafka   │
│(Idemp Key)        │  Events  │
└──────────┘        └────┬─────┘
                         │
     ┌───────────────────┼─────────────┐
     │                   │             │
     ▼                   ▼             ▼
┌──────────┐      ┌──────────┐   ┌───────────┐
│ Provider │      │Transaction   │ │ Webhook   │
│ Adapter  │      │  Worker  │   │ Service │
└────┬─────┘      └────┬─────┘   └───────────┘
     │                 │
     ▼                 ▼
┌──────────┐      ┌──────────────┐
│ Payment  │      │  PostgreSQL  │
│ Provider │      │(Transactions)│
│(Stripe,  │      └──────────────┘
│ Razorpay)│
└──────────┘

Transaction Flow (Exactly-Once):
1. Client → Payment Service with idempotency_key
2. Check Redis: if key exists → return cached result
```

```
3. Begin DB transaction:
   - INSERT into transactions (PENDING)
   - Publish to Kafka with transactional producer
   - Commit DB + Kafka atomically
4. Kafka Consumer (idempotent):
   - Read with enable.idempotence=true
   - Call payment provider
   - Update transaction status
   - Publish result event
5. Webhook Service → Notify merchant
6. Cache result in Redis (24h TTL)

Exactly-Once Semantics:
┌─────────────────────────────┐
│ Transactional Producer      │
│ (Kafka + DB atomic commit)  │
└─────────────────────────────┘
             │
             │
             ▼

┌─────────────────────────────┐
│ Idempotent Consumer         │
│ (deduplication by msg ID)   │
└─────────────────────────────┘
```

Kafka Exactly-Once Configuration:

```java
// Producer configuration
Properties props = new Properties();
props.put("enable.idempotence", "true");
props.put("transactional.id", "payment-producer-1");
props.put("acks", "all");

// Transactional send
producer.initTransactions();
try {
    producer.beginTransaction();

    // 1. Send to Kafka
    producer.send(new ProducerRecord<>("payments", txnEvent));

    // 2. Update database (within same transaction context)
    dbConnection.execute("UPDATE transactions SET status = ? WHERE id =
?");

    producer.commitTransaction();
} catch (Exception e) {
    producer.abortTransaction();
}

// Consumer configuration
props.put("isolation.level", "read_committed");
props.put("enable.auto.commit", "false");
```

Idempotency Implementation:

```
import java.time.Duration;
import java.util.concurrent.CompletableFuture;

/**
 * Payment processor with idempotency support.
 * Ensures exactly-once processing using distributed locks and caching.
 */
@Service
public class IdempotentPaymentService {

    private final RedisTemplate<String, String> redisTemplate;
    private final TransactionRepository transactionRepository;
    private final PaymentProvider paymentProvider;
    private final KafkaTemplate<String, Transaction> kafkaTemplate;

    private static final Duration LOCK_TIMEOUT = Duration.ofMinutes(5);
    private static final Duration IDEMPOTENCY_CACHE_TTL =
Duration.ofHours(24);

    @Autowired
    public IdempotentPaymentService(RedisTemplate<String, String>
redisTemplate,
                                     TransactionRepository
transactionRepository,
                                     PaymentProvider paymentProvider,
                                     KafkaTemplate<String, Transaction>
kafkaTemplate) {
        this.redisTemplate = redisTemplate;
        this.transactionRepository = transactionRepository;
        this.paymentProvider = paymentProvider;
        this.kafkaTemplate = kafkaTemplate;
    }

    @Transactional
    public CompletableFuture<PaymentResult> processPayment(PaymentRequest
request) {
        String idempotencyKey = request.getIdempotencyKey();
        String cacheKey = "idempotency:" + idempotencyKey;
        String lockKey = "lock:idempotency:" + idempotencyKey;

        // Step 1: Check if already processed (cached result)
        String cachedResult = redisTemplate.opsForValue().get(cacheKey);
        if (cachedResult != null) {
            return CompletableFuture.completedFuture(
                deserializeResult(cachedResult));
        }

        // Step 2: Acquire distributed lock
        Boolean lockAcquired = redisTemplate.opsForValue()
```

```
            .setIfAbsent(lockKey, "1", LOCK_TIMEOUT);

        if (Boolean.FALSE.equals(lockAcquired)) {
            // Another request with same key is processing - wait and
retry
            return waitAndRetry(cacheKey);
        }

        try {
            return processPaymentInternal(request, cacheKey, lockKey);
        } finally {
            // Always release lock
            redisTemplate.delete(lockKey);
        }
    }

    private CompletableFuture<PaymentResult> processPaymentInternal(
            PaymentRequest request, String cacheKey, String lockKey) {

        // Step 3: Create transaction record with PENDING status
        Transaction transaction = Transaction.builder()
            .idempotencyKey(request.getIdempotencyKey())
            .amount(request.getAmount())
            .currency(request.getCurrency())
            .status(TransactionStatus.PENDING)
            .createdAt(Instant.now())
            .build();

        transaction = transactionRepository.save(transaction);

        // Step 4: Send to Kafka (transactional)
        kafkaTemplate.executeInTransaction(operations -> {
            operations.send("payments-topic", transaction);
            return true;
        });

        // Step 5: Call payment provider
        PaymentProviderResult providerResult =
paymentProvider.charge(request);

        // Step 6: Update transaction status
        transaction.setStatus(providerResult.isSuccess()
            ? TransactionStatus.SUCCESS
            : TransactionStatus.FAILED);

transaction.setProviderTransactionId(providerResult.getTransactionId());
        transactionRepository.save(transaction);

        // Step 7: Cache result for idempotency
        PaymentResult result = PaymentResult.from(transaction,
providerResult);
        redisTemplate.opsForValue().set(
            cacheKey,
            serializeResult(result),
            IDEMPOTENCY_CACHE_TTL);

        return CompletableFuture.completedFuture(result);
    }

    private CompletableFuture<PaymentResult> waitAndRetry(String cacheKey)
{
        return CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(100);
                String cached = redisTemplate.opsForValue().get(cacheKey);
                if (cached != null) {
                    return deserializeResult(cached);
                }
                throw new PaymentProcessingException(
                    "Payment processing in progress by another request");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new PaymentProcessingException("Interrupted while
waiting");
            }
        });
    }

    private String serializeResult(PaymentResult result) {
        return new ObjectMapper().writeValueAsString(result);
    }

    private PaymentResult deserializeResult(String json) {
        return new ObjectMapper().readValue(json, PaymentResult.class);
    }
}
```

Trade-offs & Assumptions

- **Kafka vs Direct DB**: Kafka adds complexity but enables event sourcing and scalability
- **Idempotency Window**: 24h cache TTL balances storage vs retry window
- **Provider Failures**: Retry with exponential backoff (max 5 attempts), then mark as failed
- **Distributed Locks**: Redis locks prevent concurrent processing; potential bottleneck at high scale
- **Assumption**: 85% success rate; optimize for happy path
- **PCI Compliance**: Tokenize card data, never store CVV, encrypt all PII

---

9. File Storage Service

Problem Overview

Design a cloud file storage service similar to Google Drive/Dropbox, supporting file upload/download, sync across devices, sharing, and version control.

Back-of-the-Envelope Estimation

- **Users**: 100 million
- **Files per user**: Average 500 files
- **Total files**: 50 billion
- **Storage per user**: Average 10GB
- **Total storage**: 1 exabyte (1M TB)
- **Upload/download**: 10M operations/day = 116 ops/sec (peak: 1000 ops/sec)
- **Sync operations**: 100M/day = 1160 ops/sec

Functional Requirements

- **FR1**: Upload/download files (any type, up to 5GB per file)
- **FR2**: Sync files across multiple devices automatically
- **FR3**: Share files/folders with permissions (view, edit)
- **FR4**: Version history (restore previous versions)
- **FR5**: Search files by name, type, content

Non-Functional Requirements

- **Scalability**: Support 100M users, 1 exabyte storage
- **Availability**: 99.9% uptime
- **Latency**: <500ms for metadata, <5s for file download start
- **Durability**: 99.999999999% (11 nines) data durability
- **Consistency**: Strong consistency for metadata, eventual for sync

High-Level Architecture

Components:

- **Client**: Desktop/mobile sync clients
- **API Gateway**: Authentication, load balancing
- **Metadata Service**: File/folder hierarchy, permissions
- **Block Service**: Chunking, deduplication
- **Storage Service**: Object storage interface
- **Sync Service**: Push notifications for file changes
- **Share Service**: Permission management
- **Search Service**: File indexing
- **Object Storage**: S3/GCS (multiple regions)
- **Database**: PostgreSQL (metadata), Cassandra (block index)

Data Storage Choices

Data Type	Storage	Justification
File Blocks	S3/GCS	Durable object storage, 11 nines durability
Metadata	PostgreSQL	ACID, complex queries, hierarchical data
Block Index	Cassandra	Fast lookups for deduplication
User Sessions	Redis	Fast auth token validation

Data Type	Storage	Justification
Sync Queue	Redis + Pub/Sub	Real-time notifications
Search Index	Elasticsearch	Full-text search on filenames/content

Schema:

```
-- PostgreSQL (metadata)
users (
  id UUID PRIMARY KEY,
  email VARCHAR(255) UNIQUE,
  storage_used BIGINT,
  storage_quota BIGINT
)

files (
  id UUID PRIMARY KEY,
  user_id UUID,
  parent_folder_id UUID,
  name VARCHAR(255),
  size BIGINT,
  mime_type VARCHAR(100),
  version INT,
  is_deleted BOOLEAN,
  created_at TIMESTAMP,
  updated_at TIMESTAMP
)

file_versions (
  id UUID PRIMARY KEY,
  file_id UUID,
  version INT,
  size BIGINT,
  checksum VARCHAR(64),
  block_list JSONB, -- array of block hashes
  created_at TIMESTAMP
)

blocks (
  hash VARCHAR(64) PRIMARY KEY,  -- SHA-256
  size INT,
  storage_path VARCHAR(500),
  ref_count INT  -- for garbage collection
)

shares (
  id UUID PRIMARY KEY,
  file_id UUID,
  shared_with_user_id UUID,
  permission VARCHAR(20),  -- view, edit
  created_at TIMESTAMP
)
```

High-Level Diagram

```
┌──────────────┐
│    Client    │
│(Sync Daemon) │
└──────┬───────┘
       │
       ▼
┌──────────────────┐
│   API Gateway    │
│   + Auth         │
└────────┬─────────┘
         │
    ┌────┴──────────┬──────────────┬──────────────┐
    ▼               ▼              ▼              ▼
┌──────────┐   ┌──────────┐   ┌──────────┐   ┌──────────┐
│ Metadata │   │  Block   │   │   Sync   │   │  Share   │
│ Service  │   │ Service  │   │ Service  │   │ Service  │
└────┬─────┘   └────┬─────┘   └────┬─────┘   └────┬─────┘
     │              │              │              │
     ▼              ▼              │              ▼
┌──────────┐   ┌──────────┐       │        ┌──────────┐
│PostgreSQL│   │Cassandra │       │        │PostgreSQL│
│(Metadata)│   │ (Blocks) │       │        │ (Perms)  │
└──────────┘   └────┬─────┘       │        └──────────┘
                    │              │
                    ▼              ▼
               ┌──────────────────────┐
               │       S3/GCS         │
               │ (Block Storage)      │
               │ Multi-region         │
               └──────────────────────┘

              Sync Notification:
                    ┌──────────┐
                    │  Redis   │
                    │ Pub/Sub  │
                    └──────────┘

Upload Flow (Chunking + Deduplication):
1. Client: Break file into 4MB chunks
2. Client: Calculate SHA-256 for each chunk
3. Client → Block Service: Check which chunks exist
4. Block Service → Cassandra: Lookup hashes
5. Client: Upload only missing chunks → S3
6. Client → Metadata Service: Create file record
7. Metadata Service: Store block_list in file_versions
8. Sync Service: Notify other devices via Redis Pub/Sub

Download Flow:
```

```
1. Client → Metadata Service: Get file metadata
2. Metadata Service → Return block_list (array of hashes)
3. Client → Block Service: Fetch blocks by hash
4. Block Service → S3: Retrieve chunks
5. Client: Reassemble file from chunks

Chunking Strategy:
┌───────────────────────────────┐
│   File (100MB example)        │
└───────────────────────────────┘
       │
       │
       ├─► Chunk 1 (4MB) → hash1 ─┐
       ├─► Chunk 2 (4MB) → hash2  │
       ├─► Chunk 3 (4MB) → hash3  ├─► Store in S3
       ├─► ...                     │   as separate objects
       └─► Chunk 25 (4MB)→ hash25─┘

Deduplication:
- Same file uploaded by 2 users → store once
- Modified file → only upload changed chunks
- Storage savings: ~30-40% for typical workloads
```

Sync Protocol:

```java
import java.io.*;
import java.nio.file.*;
import java.security.MessageDigest;
import java.util.*;
import java.util.stream.Collectors;

/**
 * Client-side file synchronization daemon implementing chunked uploads
 * with deduplication support for efficient cloud storage sync.
 */
public class FileSyncClient {

    private static final int CHUNK_SIZE = 4 * 1024 * 1024; // 4MB chunks
    private final Path syncFolder;
    private final CloudStorageApi api;
    private final String userId;

    public FileSyncClient(Path syncFolder, CloudStorageApi api, String
userId) {
        this.syncFolder = syncFolder;
        this.api = api;
        this.userId = userId;
    }

    /**
     * Synchronize a file to the cloud storage.
     * Uses chunking and deduplication to minimize data transfer.
     */
    public void syncFile(Path filePath) throws IOException {
        // Step 1: Chunk the file
        List<byte[]> chunks = chunkFile(filePath);

        // Step 2: Calculate SHA-256 hash for each chunk
        List<String> chunkHashes = chunks.stream()
            .map(this::calculateSha256)
            .collect(Collectors.toList());

        // Step 3: Check which chunks already exist on server
        CheckChunksResponse response = api.checkChunks(chunkHashes);
        Set<String> missingHashes = new HashSet<>
(response.getMissingHashes());

        // Step 4: Upload only missing chunks (deduplication)
        for (int i = 0; i < chunkHashes.size(); i++) {
            String hash = chunkHashes.get(i);
            if (missingHashes.contains(hash)) {
                api.uploadChunk(hash, chunks.get(i));
            }
        }

        // Step 5: Create file metadata on server
        long totalSize = chunks.stream().mapToLong(c -> c.length).sum();
        api.createFile(CreateFileRequest.builder()
            .name(filePath.getFileName().toString())
            .size(totalSize)
            .blocks(chunkHashes)
            .build());
    }

    /**
     * Split file into fixed-size chunks for parallel upload
     * and deduplication.
     */
    private List<byte[]> chunkFile(Path filePath) throws IOException {
        List<byte[]> chunks = new ArrayList<>();

        try (InputStream inputStream = Files.newInputStream(filePath);
             BufferedInputStream buffered = new
BufferedInputStream(inputStream)) {

            byte[] buffer = new byte[CHUNK_SIZE];
            int bytesRead;

            while ((bytesRead = buffered.read(buffer)) != -1) {
                byte[] chunk = (bytesRead == CHUNK_SIZE)
                    ? buffer.clone()
                    : Arrays.copyOf(buffer, bytesRead);
                chunks.add(chunk);
            }
        }

        return chunks;
    }

    /**
     * Compute SHA-256 hash for content-addressable storage.
     */
    private String calculateSha256(byte[] data) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(data);
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                hexString.append(String.format("%02x", b));
            }
            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException("SHA-256 calculation failed", e);
        }
    }

    /**
     * Watch for file system changes and trigger sync.
     * Uses Java WatchService for efficient event-driven detection.
     */
    public void startWatching() throws IOException {
        WatchService watchService =
FileSystems.getDefault().newWatchService();
        syncFolder.register(watchService,
            StandardWatchEventKinds.ENTRY_CREATE,
            StandardWatchEventKinds.ENTRY_MODIFY,
            StandardWatchEventKinds.ENTRY_DELETE);

        // Start server notification listener in separate thread
        Thread notificationListener = new
Thread(this::listenForServerChanges);
        notificationListener.setDaemon(true);
        notificationListener.start();

        // Process file system events
        while (true) {
            try {
                WatchKey key = watchService.take();

                for (WatchEvent<?> event : key.pollEvents()) {
                    WatchEvent.Kind<?> kind = event.kind();
                    Path fileName = (Path) event.context();
                    Path fullPath = syncFolder.resolve(fileName);

                    if (kind == StandardWatchEventKinds.ENTRY_CREATE ||
                        kind == StandardWatchEventKinds.ENTRY_MODIFY) {
                        syncFile(fullPath);
                    } else if (kind ==
StandardWatchEventKinds.ENTRY_DELETE) {
                        api.deleteFile(fileName.toString());
                    }
                }

                key.reset();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }

    /**
     * Subscribe to server-side changes via Redis Pub/Sub
     * for real-time bi-directional sync.
     */
    private void listenForServerChanges() {
        try (Jedis jedis = new Jedis("redis://localhost:6379")) {
            jedis.subscribe(new JedisPubSub() {
                @Override
                public void onMessage(String channel, String message) {
                    try {
                        FileChangeEvent event = parseChangeEvent(message);
                        downloadFile(event.getFileId());
                    } catch (IOException e) {
                        System.err.println("Error downloading file: " +
e.getMessage());
                    }
                }
            }, "user:" + userId + ":changes");
        }
    }

    private void downloadFile(String fileId) throws IOException {
        // Download file implementation
        FileMetadata metadata = api.getFileMetadata(fileId);
        List<byte[]> chunks = new ArrayList<>();

        for (String blockHash : metadata.getBlocks()) {
            byte[] chunk = api.downloadChunk(blockHash);
            chunks.add(chunk);
        }

        // Reassemble file
        Path targetPath = syncFolder.resolve(metadata.getName());
        try (OutputStream out = Files.newOutputStream(targetPath)) {
            for (byte[] chunk : chunks) {
                out.write(chunk);
            }
        }
    }

    private FileChangeEvent parseChangeEvent(String message) {
        // JSON parsing implementation
        return new ObjectMapper().readValue(message,
FileChangeEvent.class);
```

```
    }
}
```

Trade-offs & Assumptions

- **Chunking**: 4MB chunks balance deduplication vs overhead; smaller chunks = more metadata
- **Deduplication**: Block-level saves storage but adds complexity; file-level simpler but less effective
- **Sync Strategy**: Push notifications via Pub/Sub vs polling; push is real-time but requires persistent connections
- **Versioning**: Keep last 30 versions; older versions moved to Glacier
- **Assumption**: 70% of data is duplicate (office docs, media); deduplication provides major savings
- **Consistency**: Metadata updates use transactions; last-write-wins for concurrent edits (conflict resolution needed)

---

10. Flight Booking System

Problem Overview

Design a flight booking system handling seat reservations with concurrent booking contention, payment processing, failure recovery, and synchronization with external aggregators (MakeMyTrip, Booking.com).

Back-of-the-Envelope Estimation

- **Flights/day**: 100,000 flights worldwide
- **Seats/flight**: Average 200 seats
- **Total inventory**: 20 million seats/day
- **Bookings/day**: 5 million (25% load factor)
- **Peak bookings/sec**: 5M / 86400 × 10 (peak) = ~580 bookings/sec
- **Concurrent users**: 1 million
- **Aggregator sync**: 100 aggregators × 1000 flights each = 100K updates/min

Functional Requirements

- **FR1**: Search flights by route, date, passengers
- **FR2**: Select seats and hold temporarily (5-10 min hold)
- **FR3**: Complete booking with payment
- **FR4**: Handle payment failures and retry
- **FR5**: Sync inventory with external aggregators in real-time

Non-Functional Requirements

- **Scalability**: Handle 580 bookings/sec peak load
- **Availability**: 99.95% uptime for bookings
- **Latency**: <200ms for search, <3s for booking
- **Consistency**: Strong consistency for seat inventory (no double bookings)
- **Atomicity**: Booking + payment atomic transaction
- **Sync Latency**: Update aggregators within 5 seconds

High-Level Architecture

Components:

- **Client**: Web/Mobile apps
- **API Gateway**: Load balancing, rate limiting
- **Search Service**: Flight availability queries
- **Booking Service**: Reservation orchestration
- **Inventory Service**: Seat availability management
- **Payment Service**: Payment processing
- **Lock Service**: Distributed locking (Redis/etcd)
- **Aggregator Sync Service**: Push updates to partners
- **Database**: PostgreSQL (bookings), Redis (inventory cache)
- **Message Queue**: Kafka (event streaming)

Data Storage Choices

Data Type	Storage	Justification
Flight Inventory	PostgreSQL + Redis	Strong consistency with caching
Bookings	PostgreSQL	ACID transactions
Seat Locks	Redis	Fast TTL-based locking
Payment Transactions	PostgreSQL	Audit trail, ACID
Sync Queue	Kafka	Reliable aggregator updates
Session State	Redis	Temporary booking holds

Schema:

```
-- PostgreSQL
flights (
  id BIGINT PRIMARY KEY,
  flight_number VARCHAR(10),
  route VARCHAR(100),
  departure_time TIMESTAMP,
  arrival_time TIMESTAMP,
  total_seats INT,
  available_seats INT,
  version INT  -- optimistic locking
)

seats (
  id BIGINT PRIMARY KEY,
  flight_id BIGINT,
  seat_number VARCHAR(5),
  class VARCHAR(20),
  status VARCHAR(20),  -- available, held, booked
  price DECIMAL(10,2),
  held_until TIMESTAMP,
  held_by_session VARCHAR(100)
)

bookings (
  id UUID PRIMARY KEY,
  user_id UUID,
  flight_id BIGINT,
  seat_ids JSONB,
  status VARCHAR(20),  -- pending, confirmed, cancelled, failed
  payment_id UUID,
  total_amount DECIMAL(10,2),
  created_at TIMESTAMP,
  confirmed_at TIMESTAMP
)

CREATE INDEX idx_seats_flight ON seats(flight_id, status);
CREATE INDEX idx_flight_version ON flights(id, version);
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌─────────────┐
│ API Gateway │
└──────┬──────┘
       │
   ┌───┴────────────┬──────────────┬──────────────┐
   ▼                ▼              ▼              ▼
┌────────┐    ┌──────────┐  ┌──────────┐  ┌──────────┐
│ Search │    │ Booking  │  │Inventory │  │ Payment  │
│Service │    │ Service  │  │ Service  │  │ Service  │
└───┬────┘    └────┬─────┘  └────┬─────┘  └────┬─────┘
    │              │             │             │
    │         ┌────▼─────┐       │             │
    │         │  Redis   │       │             │
    │         │(Seat Lock)       │             │
    │         └──────────┘       │             │
    │                            │             │
    │                            │             │
    └────────────┬───────────────┴─────────────┘
                 ▼
          ┌─────────────┐
          │ PostgreSQL  │
          │(Flights,    │
          │ Bookings)   │
          └──────┬──────┘
                 │
                 ▼
          ┌─────────────┐
          │    Kafka    │
          │ (Sync Event)│
          └──────┬──────┘
                 │
                 ▼
          ┌─────────────┐
          │ Aggregator  │
          │Sync Service │
          └──────┬──────┘
                 │
      ┌──────────┴──────────┐
      ▼                     ▼
┌──────────┐         ┌──────────┐
│MakeMyTrip│   ...   │Booking.com
└──────────┘         └──────────┘

Booking Flow (Pessimistic Locking):
1. User selects seat
2. Booking Service → Redis: Acquire lock
   SET seat:123:lock user_session_id NX EX 600
3. If lock acquired:
   a. Update seat status to 'held'
   b. Set held_until = NOW() + 10 min
   c. Return to user (10 min to complete payment)
4. User completes payment
5. Booking Service:
   BEGIN TRANSACTION
     - Insert booking record
     - Update seat status to 'booked'
     - Decrease flight available_seats
     - Commit payment
   COMMIT TRANSACTION
6. Release Redis lock
7. Publish to Kafka → Sync aggregators

Optimistic Locking (Alternative):
UPDATE flights
SET available_seats = available_seats - 1,
    version = version + 1
WHERE id = ? AND version = ? AND available_seats > 0

If affected_rows = 0 → Concurrent update detected → Retry

Double Booking Prevention:
┌─────────────────────────────────┐
│ Distributed Lock (Redis)        │
│ + Database UNIQUE constraint    │
│ + Optimistic locking (version)  │
└─────────────────────────────────┘


Payment Flow:
1. Hold seat (10 min)
2. User enters payment details
```

```
3. Payment Service:
   - Call payment gateway
   - If success → confirm booking
   - If failure → retry 2x with backoff
   - If max retries → release seat, notify user
4. Saga pattern for rollback:
   Payment failed → Undo seat booking → Release lock
```

Distributed Lock Implementation:

```java
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 * Distributed lock implementation using Redis for seat booking.
 * Supports lock acquisition, release, and extension with ownership
verification.
 */
public class DistributedLock {

    private final RedisTemplate<String, String> redisTemplate;

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

    // Lua script for atomic extend (only if we own the lock)
    private static final String EXTEND_SCRIPT = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("PEXPIRE", KEYS[1], ARGV[2])
        else
            return 0
        end
        """;

    public DistributedLock(RedisTemplate<String, String> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    /**
     * Attempt to acquire a lock for a seat with automatic expiry.
     *
     * @param seatId The seat identifier to lock
     * @param sessionId Unique session identifier (lock owner)
     * @param ttlSeconds Time-to-live for the lock
     * @return true if lock acquired, false otherwise
     */
    public boolean acquireSeatLock(String seatId, String sessionId, long
ttlSeconds) {
        String lockKey = "seat:" + seatId + ":lock";
        Boolean result = redisTemplate.opsForValue()
            .setIfAbsent(lockKey, sessionId, ttlSeconds,
TimeUnit.SECONDS);
        return Boolean.TRUE.equals(result);
    }

    /**
     * Release a lock only if the caller owns it.
     * Uses Lua script for atomic check-and-delete operation.
     */
    public void releaseSeatLock(String seatId, String sessionId) {
        String lockKey = "seat:" + seatId + ":lock";
        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(RELEASE_SCRIPT, Long.class);
        redisTemplate.execute(script, Collections.singletonList(lockKey),
sessionId);
    }

    /**
     * Extend the lock TTL if still owned by the caller.
     * Useful for long-running operations.
     */
    public boolean extendLock(String seatId, String sessionId, long
ttlMillis) {
        String lockKey = "seat:" + seatId + ":lock";
        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(EXTEND_SCRIPT, Long.class);
        Long result = redisTemplate.execute(
            script,
            Collections.singletonList(lockKey),
            sessionId,
            String.valueOf(ttlMillis));
        return result != null && result == 1;
    }
}

/**
 * Flight booking service with distributed locking and saga pattern
 * for handling seat reservations with payment processing.
 */
@Service
@Transactional
public class FlightBookingService {

    private final DistributedLock distributedLock;
    private final SeatRepository seatRepository;
    private final BookingRepository bookingRepository;
    private final FlightRepository flightRepository;
    private final PaymentService paymentService;
    private final KafkaTemplate<String, InventoryUpdate> kafkaTemplate;

    private static final long SEAT_HOLD_TTL_SECONDS = 600; // 10 minutes

    @Autowired
    public FlightBookingService(DistributedLock distributedLock,
                                 SeatRepository seatRepository,
                                 BookingRepository bookingRepository,
                                 FlightRepository flightRepository,
                                 PaymentService paymentService,
                                 KafkaTemplate<String, InventoryUpdate>
kafkaTemplate) {
        this.distributedLock = distributedLock;
        this.seatRepository = seatRepository;
        this.bookingRepository = bookingRepository;
        this.flightRepository = flightRepository;
        this.paymentService = paymentService;
        this.kafkaTemplate = kafkaTemplate;
    }

    /**
     * Hold a seat temporarily while user completes payment.
     * Returns a booking hold valid for 10 minutes.
     */
    public SeatHoldResponse holdSeat(UUID userId, Long flightId,
                                      Long seatId, String sessionId) {
        // Step 1: Acquire distributed lock
        if (!distributedLock.acquireSeatLock(seatId.toString(),
                                             sessionId,
                                             SEAT_HOLD_TTL_SECONDS)) {
            throw new SeatAlreadyHeldException("Seat is currently held by
another user");
        }

        try {
            // Step 2: Update seat status in database
            int rowsUpdated = seatRepository.holdSeat(
                seatId,
                sessionId,
                Instant.now().plusSeconds(SEAT_HOLD_TTL_SECONDS));

            if (rowsUpdated == 0) {
                throw new SeatNotAvailableException("Seat is no longer
available");
            }

            return new SeatHoldResponse(
                seatId,
                Instant.now().plusSeconds(SEAT_HOLD_TTL_SECONDS));

        } catch (Exception e) {
            // Release lock on any error
            distributedLock.releaseSeatLock(seatId.toString(), sessionId);
            throw e;
        }
    }

    /**
     * Confirm a booking by processing payment and finalizing the
reservation.
     * Implements saga pattern with compensating transactions for
failures.
     */
    public Booking confirmBooking(UUID bookingId, PaymentDetails
paymentDetails) {
        Booking booking = bookingRepository.findById(bookingId)
            .orElseThrow(() -> new BookingNotFoundException("Booking not
found"));

        // Idempotency check
        if (booking.getStatus() == BookingStatus.CONFIRMED) {
            return booking;
        }

        String sessionId = booking.getSessionId();
        List<Long> seatIds = booking.getSeatIds();

        try {
            // Step 1: Process payment
            PaymentResult paymentResult =
paymentService.charge(paymentDetails);

            // Step 2: Confirm booking atomically
            bookingRepository.confirmBooking(
                bookingId,
                paymentResult.getPaymentId(),
                Instant.now());

            // Step 3: Update seat status to booked
            seatRepository.confirmBooking(seatIds);

            // Step 4: Decrement flight available seats
            flightRepository.decrementAvailableSeats(
                booking.getFlightId(),
                seatIds.size());

            // Step 5: Publish inventory update to aggregators
            kafkaTemplate.send("inventory-updates",
InventoryUpdate.builder()
                .flightId(booking.getFlightId())
                .seatsBooked(seatIds)
                .timestamp(Instant.now())
                .build());

            // Step 6: Release distributed lock
            for (Long seatId : seatIds) {
                distributedLock.releaseSeatLock(seatId.toString(),
sessionId);
```

```
            }

            booking.setStatus(BookingStatus.CONFIRMED);
            return booking;

        } catch (PaymentException e) {
            // Compensating transaction: release held seats
            executeCompensation(seatIds, sessionId);
            throw new BookingFailedException("Payment failed: " +
e.getMessage(), e);
        }
    }

    /**
     * Execute compensating transaction to release seats on failure.
     */
    private void executeCompensation(List<Long> seatIds, String sessionId)
{
        seatRepository.releaseSeatsByIds(seatIds);
        for (Long seatId : seatIds) {
            distributedLock.releaseSeatLock(seatId.toString(), sessionId);
        }
    }
}

/**
 * Response DTO for seat hold operation.
 */
public record SeatHoldResponse(Long seatId, Instant heldUntil) {}

/**
 * Builder pattern for inventory updates sent to external aggregators.
 */
@Builder
@Data
public class InventoryUpdate {
    private Long flightId;
    private List<Long> seatsBooked;
    private Instant timestamp;
}
```

Trade-offs & Assumptions

- **Pessimistic vs Optimistic Locking**: Pessimistic prevents contention but requires lock management; use for high-demand flights
- **Lock TTL**: 10 min balance between user experience and inventory blocking
- **Payment Retry**: Max 2 retries to avoid long delays; user can re-attempt booking
- **Aggregator Sync**: Async via Kafka; eventual consistency acceptable (5-10 sec delay)
- **Assumption**: 5% of flights have high contention (>50% booking rate); rest can use simpler locking
- **Database Isolation**: Use SERIALIZABLE for critical sections; performance cost acceptable for correctness

---

11. Flight Price Management System

Problem Overview

Design a system to manage and retrieve flight prices from multiple providers, handling per-provider rate limiting and distributed datacenter challenges with price synchronization.

Back-of-the-Envelope Estimation

- **Providers**: 50 airlines + aggregators
- **Flight routes**: 100K unique routes
- **Price updates/day**: 10M updates (prices change frequently)
- **Query rate**: 5000 queries/sec (peak)
- **Per-provider rate limit**: 100 req/sec
- **Datacenters**: 3 regions (US, EU, APAC)

Functional Requirements

- **FR1**: Fetch prices from multiple providers with rate limiting
- **FR2**: Cache prices with configurable TTL (2-30 min)
- **FR3**: Aggregate prices and find cheapest option
- **FR4**: Handle provider outages gracefully
- **FR5**: Sync prices across distributed datacenters

Non-Functional Requirements

- **Scalability**: Support 50 providers, 5000 queries/sec
- **Availability**: 99.9% uptime
- **Latency**: <500ms for price retrieval
- **Consistency**: Eventual consistency across DCs (acceptable delay: 30 seconds)
- **Cost**: Minimize API calls through intelligent caching

High-Level Architecture

Components:

- **API Gateway**: Per-DC entry point
- **Price Service**: Query orchestration
- **Provider Gateway**: Rate limiting per provider
- **Cache Layer**: Redis (multi-level)
- **Price Aggregator**: Background price updates
- **Sync Service**: Cross-DC replication
- **Database**: Cassandra (price history)

Data Storage Choices

Data Type	Storage	Justification

Data Type	Storage	Justification
Current Prices	Redis (per-DC)	Fast access, TTL support, sub-ms latency
Price History	Cassandra	Time-series data, multi-DC replication
Provider Config	PostgreSQL	Rate limits, credentials, routing
Cache Stats	ClickHouse	Analytics on hit rates, costs

Schema (Cassandra):

```
CREATE TABLE price_snapshots (
  route_id UUID,
  provider VARCHAR,
  timestamp TIMESTAMP,
  price DECIMAL,
  currency VARCHAR,
  availability INT,
  PRIMARY KEY ((route_id, provider), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

High-Level Diagram

```
          Datacenter US              Datacenter EU              Datacenter
APAC
     ┌─────────────────┐         ┌─────────────────┐
┌─────────────────┐
     │  API Gateway    │         │  API Gateway    │         │  API
Gateway     │
     └────────┬────────┘         └────────┬────────┘
└────────┬────────┘
              │                           │                           │
              ▼                           ▼                           ▼
     ┌─────────────────┐         ┌─────────────────┐
┌─────────────────┐
     │ Price Service   │◄────────│ Price Service   │◄────────│ Price
Service    │
     └────────┬────────┘  Sync   └────────┬────────┘  Sync
└────────┬────────┘
              │                           │                           │
              ▼                           ▼                           ▼
     ┌─────────────────┐         ┌─────────────────┐
┌─────────────────┐
     │  Redis Cache    │         │  Redis Cache    │         │  Redis
Cache    │
     │  (Local prices) │         │  (Local prices) │         │  (Local
prices) │
     └────────┬────────┘         └────────┬────────┘
└────────┬────────┘
              │                           │                           │
```

```
                 └────────────────────────────┼────────────────────────────┘
                                              │
                                              ▼
                                   ┌─────────────────┐
                                   │    Cassandra    │
                                   │  (Multi-DC      │
                                   │   Replication)  │
                                   └─────────────────┘

Provider Gateway (Rate Limiting):
┌───────────────────────────────────┐
│   Token Bucket per Provider       │
│   Provider A: 100 req/sec         │
│   Provider B: 50 req/sec          │
│   ...                             │
└───────────────────────────────────┘
         │
         │
    ┌────┴──────┬──────────┬────────┐
    │           │          │        │
    ▼           ▼          ▼        ▼
┌────────┐  ┌────────┐ ┌────────┐
│API #1  │  │API #2  │ │API #3  │
└────────┘  └────────┘ └────────┘

Query Flow:
1. User → API Gateway → Price Service
2. Price Service → Check Redis cache
3. If cache miss or stale:
   - Query Provider Gateway
   - Provider Gateway: Apply rate limit
   - If within limit → Call provider API
   - If rate limited → Return cached (stale) or next provider
4. Aggregate results from multiple providers
5. Update cache with new prices
6. Async: Sync to other DCs via Kafka

Rate Limiting (Token Bucket):
class ProviderRateLimiter:
    def __init__(self, rate=100, capacity=100):
        self.rate = rate  # tokens per second
        self.capacity = capacity
        self.tokens = capacity
        self.last_update = time.time()

    def acquire(self):
        now = time.time()
        elapsed = now - self.last_update
        self.tokens = min(self.capacity, self.tokens + elapsed *
self.rate)
        self.last_update = now

        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False
```

Cross-DC Synchronization:

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.data.redis.core.RedisTemplate;
import com.datastax.oss.driver.api.core.CqlSession;
import java.time.*;
import java.util.concurrent.*;

/**
 * Service for managing flight prices with cross-datacenter
synchronization.
 * Maintains local Redis cache with Cassandra as persistent store and
 * Kafka for inter-DC event propagation.
 */
@Service
public class PriceSyncService {

    private final RedisTemplate<String, String> redisTemplate;
    private final CqlSession cassandraSession;
    private final KafkaTemplate<String, PriceUpdateEvent> kafkaTemplate;
    private final String currentDatacenter;

    private static final Duration CACHE_TTL = Duration.ofMinutes(5);
    private static final String PRICE_UPDATE_TOPIC = "price-updates";

    @Autowired
    public PriceSyncService(RedisTemplate<String, String> redisTemplate,
                            CqlSession cassandraSession,
                            KafkaTemplate<String, PriceUpdateEvent>
kafkaTemplate,
                            @Value("${datacenter.id}") String
currentDatacenter) {
        this.redisTemplate = redisTemplate;
        this.cassandraSession = cassandraSession;
        this.kafkaTemplate = kafkaTemplate;
        this.currentDatacenter = currentDatacenter;
    }

    /**
     * Update flight price in local cache, persistent store, and propagate
to other DCs.
     * Implements eventual consistency across datacenters.
     */
    public CompletableFuture<Void> updatePrice(String routeId, String
provider,
                                               Price price) {
        String cacheKey = String.format("price:%s:%s", routeId, provider);

        // Step 1: Update local Redis cache immediately for low latency
        String priceJson = serializePrice(price);
        redisTemplate.opsForValue().set(cacheKey, priceJson, CACHE_TTL);
        // Step 2: Persist to Cassandra (multi-DC replication handled by
Cassandra)
        return CompletableFuture.runAsync(() -> {
            persistToCassandra(routeId, provider, price);
        }).thenCompose(v -> {
            // Step 3: Publish to Kafka for other DCs to update their
local caches
            PriceUpdateEvent event = PriceUpdateEvent.builder()
                .routeId(routeId)
                .provider(provider)
                .price(price)
                .datacenter(currentDatacenter)
                .timestamp(Instant.now())
                .build();

            return kafkaTemplate.send(PRICE_UPDATE_TOPIC, routeId, event)
                .thenApply(result -> null);
        });
    }

    private void persistToCassandra(String routeId, String provider, Price
price) {
        String insertQuery = """
            INSERT INTO price_snapshots
            (route_id, provider, timestamp, price, currency)
            VALUES (?, ?, ?, ?, ?)
            """;

        cassandraSession.execute(
            cassandraSession.prepare(insertQuery).bind(
                routeId,
                provider,
                Instant.now(),
                price.getAmount(),
                price.getCurrency()
            ));
    }

    private String serializePrice(Price price) {
        try {
            return new ObjectMapper().writeValueAsString(price);
        } catch (JsonProcessingException e) {
            throw new RuntimeException("Failed to serialize price", e);
        }
    }
}

/**
 * Kafka consumer for processing price updates from other datacenters.
 * Updates local Redis cache to maintain cross-DC consistency.
 */
@Component
public class PriceUpdateConsumer {
    private final RedisTemplate<String, String> redisTemplate;
    private final String currentDatacenter;

    private static final Duration CACHE_TTL = Duration.ofMinutes(5);

    @Autowired
    public PriceUpdateConsumer(RedisTemplate<String, String>
redisTemplate,
                               @Value("${datacenter.id}") String
currentDatacenter) {
        this.redisTemplate = redisTemplate;
        this.currentDatacenter = currentDatacenter;
    }

    /**
     * Consume price updates from other DCs and update local cache.
     * Ignores events originating from this datacenter (already processed
locally).
     */
    @KafkaListener(topics = "price-updates", groupId = "${datacenter.id}-
price-consumer")
    public void consumePriceUpdate(PriceUpdateEvent event) {
        // Skip events from this datacenter (already updated locally)
        if (currentDatacenter.equals(event.getDatacenter())) {
            return;
        }

        String cacheKey = String.format("price:%s:%s",
            event.getRouteId(), event.getProvider());

        String priceJson = serializePrice(event.getPrice());
        redisTemplate.opsForValue().set(cacheKey, priceJson, CACHE_TTL);
    }

    private String serializePrice(Price price) {
        try {
            return new ObjectMapper().writeValueAsString(price);
        } catch (JsonProcessingException e) {
            throw new RuntimeException("Failed to serialize price", e);
        }
    }
}

/**
 * Event representing a price update for cross-DC propagation.
 */
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class PriceUpdateEvent {
    private String routeId;
    private String provider;
    private Price price;
    private String datacenter;
    private Instant timestamp;
}

/**
 * Value object representing a flight price.
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Price {
    private BigDecimal amount;
    private String currency;
}
```

Trade-offs & Assumptions

- **Cache TTL**: 5 min for popular routes, 30 min for others; balance freshness vs API cost
- **Multi-DC**: Each DC has local cache; improves latency but eventual consistency
- **Rate Limiting**: Per-provider limits prevent API overage charges; queue requests if needed
- **Stale Data**: Serve stale prices if provider is rate-limited; better than no data
- **Assumption**: 80% cache hit rate reduces provider API calls by 5x

---

12. Location Sharing App

Problem Overview

Design a location sharing application with granular controls allowing users to share their location with specific contacts for limited time periods and within specific geographic boundaries.

Back-of-the-Envelope Estimation

- **DAU**: 20 million users
- **Active sharing sessions**: 5M concurrent
- **Location updates**: Every 30 seconds = 167K updates/sec
- **Database writes**: 167K writes/sec
- **Query load**: 10M queries/min for shared locations = 167K reads/sec
- **Storage**: 5M sessions × 1KB = 5GB (active), 100GB/day (history)

Functional Requirements

- **FR1**: Share location with specific users (contacts)
- **FR2**: Set time-based expiry (1 hour, 8 hours, 24 hours, until cancelled)
- **FR3**: Set geographic boundary (only share if within radius)
- **FR4**: Real-time location updates (30-60 second intervals)
- **FR5**: View shared locations on map

Non-Functional Requirements

- **Scalability**: Handle 20M DAU, 167K updates/sec
- **Availability**: 99.9% uptime
- **Latency**: <500ms for location retrieval, <1s for updates
- **Privacy**: Strong access controls, encrypted location data
- **Battery Efficiency**: Minimize mobile battery drain

High-Level Architecture

Components:

- **Client**: Mobile apps with background location tracking
- **API Gateway**: Authentication, rate limiting
- **Location Service**: Location update processing
- **Sharing Service**: Permission management
- **Geo-fence Service**: Boundary validation
- **Real-time Service**: WebSocket/SSE for live updates
- **Database**: Cassandra (location history), Redis (active sessions)
- **Message Queue**: Kafka (location stream)

Data Storage Choices

Data Type	Storage	Justification
Active Locations	Redis + Geospatial	Fast geo-queries, TTL support
Location History	Cassandra	Time-series data, high write throughput
Sharing Permissions	PostgreSQL	Complex ACLs, strong consistency
User Sessions	Redis	Fast lookup, automatic expiry

Schema:

```
-- PostgreSQL
sharing_permissions (
  id UUID PRIMARY KEY,
  owner_user_id UUID,
  shared_with_user_id UUID,
  expiry_time TIMESTAMP,
  geo_fence_enabled BOOLEAN,
  geo_fence_center POINT, -- lat, lon
  geo_fence_radius_meters INT,
  created_at TIMESTAMP,
  UNIQUE(owner_user_id, shared_with_user_id)
)

CREATE INDEX idx_sharing_expiry ON sharing_permissions(expiry_time)
WHERE expiry_time > NOW();

-- Cassandra
location_updates (
  user_id UUID,
  timestamp TIMESTAMP,
  latitude DECIMAL(10,8),
  longitude DECIMAL(11,8),
  accuracy INT,
  battery_level INT,
  PRIMARY KEY (user_id, timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- Redis Geospatial
GEOADD active:locations longitude latitude user_id
```

High-Level Diagram

```
   ┌─────────────┐
   │ Mobile App  │
   │(Background  │
   │  Location)  │
   └──────┬──────┘
          │ Every 30s
          ▼
   ┌──────────────┐
   │ API Gateway  │
   └──────┬───────┘
          │
      ┌───┴──────────────┬──────────────┐
      ▼                  ▼              ▼
   ┌────────────┐   ┌──────────┐  ┌───────────┐
   │  Location  │   │ Sharing  │  │ Geo-fence │
   │  Service   │   │ Service  │  │ Service   │
   └─────┬──────┘   └────┬─────┘  └─────┬─────┘
         │               │              │
         ▼               ▼              │
   ┌───────────┐  ┌──────────┐         │
   │   Redis   │  │PostgreSQL│         │
   │ Geospatial│  │  (ACL)   │         │
   └─────┬─────┘  └──────────┘         │
         │                             │
         ▼                             │
   ┌───────────┐                       │
   │  Kafka    │◄──────────────────────┘
   │ (Stream)  │
   └─────┬─────┘
         │
         ▼
   ┌───────────┐
   │ Cassandra │
   │ (History) │
   └───────────┘

Location Update Flow:
```

```
1. App → Location Service: {user_id, lat, lon, timestamp}
2. Location Service:
   a. Validate sharing permissions
   b. Check geo-fence constraints
   c. Update Redis GEOADD
   d. Publish to Kafka
   e. Cassandra async write
3. Real-time Service:
   - Subscribe to Kafka
   - Push to connected clients via WebSocket

Geo-fence Validation:
def is_within_geofence(user_location, sharing_config):
    if not sharing_config.geo_fence_enabled:
        return True

    distance = haversine(
        user_location.lat, user_location.lon,
        sharing_config.center.lat, sharing_config.center.lon
    )

    return distance <= sharing_config.radius_meters

Query Shared Locations:
1. User A queries → "Show me all shared locations"
2. Sharing Service:
   SELECT shared_with_user_id
   FROM sharing_permissions
   WHERE owner_user_id = ? AND expiry_time > NOW()
3. For each shared user:
   GEOPOS active:locations user_id
4. Return locations with user metadata
```

Redis Geospatial Commands:

```
# Add location
GEOADD active:locations -122.4194 37.7749 user:123

# Get location
GEOPOS active:locations user:123

# Find nearby users (within 5km)
GEORADIUS active:locations -122.4194 37.7749 5 km WITHDIST

# Distance between two users
GEODIST active:locations user:123 user:456 km

# Set expiry on location
EXPIRE active:locations:user:123 3600  # 1 hour
```

WebSocket Real-time Updates:

```
// Server-side
class LocationRealtimeService {
  constructor() {
    this.connections = new Map(); // user_id -> WebSocket[]
  }

  async onConnect(ws, user_id) {
    if (!this.connections.has(user_id)) {
      this.connections.set(user_id, []);
    }
    this.connections.get(user_id).push(ws);

    // Subscribe to Kafka topic for this user's shared contacts
    const contacts = await this.getSharedContacts(user_id);
    await kafka.subscribe(`locations:${contacts.join(',')}`);
  }

  async onLocationUpdate(user_id, location) {
    // Find all users who have access to this user's location
    const subscribers = await this.getSubscribers(user_id);

    for (const subscriber of subscribers) {
      const sockets = this.connections.get(subscriber) || [];
      for (const ws of sockets) {
        ws.send(JSON.stringify({
          type: 'location_update',
          user_id: user_id,
          location: location,
          timestamp: Date.now()
        }));
      }
    }
  }
}
```

Trade-offs & Assumptions

- **Update Frequency**: 30s interval balances real-time vs battery/bandwidth
- **Geo-fence**: Client-side validation first, server-side enforcement; prevents unnecessary updates
- **Redis TTL**: 1 hour for active locations; auto-cleanup for expired sessions
- **WebSocket vs Polling**: WebSocket for real-time, fallback to polling for poor connections
- **Assumption**: Average 10 sharing relationships per user; 90% of shares are time-limited (<24h)
- **Privacy**: End-to-end encryption option for high-security use cases

---

13. WhatsApp

Problem Overview

Design a messaging platform like WhatsApp supporting real-time one-to-one and group messaging, media sharing, end-to-end encryption, read receipts, and offline message delivery.

Back-of-the-Envelope Estimation

- **DAU**: 2 billion users
- **Messages/day**: 100 billion
- **Messages/sec**: 100B / 86400 = 1.16M messages/sec (peak: 5M msg/sec)
- **Media messages**: 30% of total = 30B files/day
- **Group messages**: 40% of total, avg group size: 10
- **Storage**: 100B × 1KB avg = 100TB/day metadata, 30B × 500KB = 15PB/day media
- **Online users**: 500M concurrent

Functional Requirements

- **FR1**: Send/receive one-to-one messages in real-time
- **FR2**: Create groups and send group messages
- **FR3**: Send media files (images, videos, documents)
- **FR4**: End-to-end encryption for all messages
- **FR5**: Delivery and read receipts
- **FR6**: Offline message delivery (store and forward)
- **FR7**: Last seen and online status

Non-Functional Requirements

- **Scalability**: Support 2B users, 5M messages/sec
- **Availability**: 99.99% uptime
- **Latency**: <200ms message delivery (same region)
- **Consistency**: At-least-once delivery, ordered within conversation
- **Privacy**: E2E encryption, metadata minimization
- **Storage**: Efficient media storage with deduplication

High-Level Architecture

Components:

- **Client**: Mobile/Desktop apps with local encryption
- **Gateway**: WebSocket connections (persistent)
- **Message Router**: Route messages to recipients
- **Message Storage**: Temporary storage for offline users
- **Media Service**: Upload/download media files
- **User Service**: Contacts, profile, online status
- **Group Service**: Group membership management
- **Notification Service**: Push notifications for offline users
- **Database**: Cassandra (messages), PostgreSQL (users), S3 (media)
- **Cache**: Redis (online status, message buffer)

Data Storage Choices

Data Type	Storage	Justification
Messages (7-30 days)	Cassandra	Time-series, high write throughput, partition by user
Media Files	S3 + CDN	Blob storage, global distribution
User Profiles	PostgreSQL	Relational data, complex queries
Online Status	Redis	Fast reads/writes, TTL
Message Queue	Kafka	Durable buffer for offline messages
Group Metadata	PostgreSQL	ACID for membership changes

Schema:

```
-- PostgreSQL
users (
  id UUID PRIMARY KEY,
  phone_number VARCHAR(20) UNIQUE,
  username VARCHAR(50),
  profile_photo_url VARCHAR(500),
  created_at TIMESTAMP,
  last_seen TIMESTAMP
)

groups (
  id UUID PRIMARY KEY,
  name VARCHAR(255),
  created_by UUID,
  created_at TIMESTAMP
)

group_members (
  group_id UUID,
  user_id UUID,
  role VARCHAR(20), -- admin, member
  joined_at TIMESTAMP,
  PRIMARY KEY (group_id, user_id)
)

-- Cassandra
messages (
  conversation_id UUID,  -- hash(sender_id, recipient_id) for 1:1
  message_id TIMEUUID,
  sender_id UUID,
  recipient_id UUID,
  content BLOB,  -- encrypted
  media_url VARCHAR(500),
  status VARCHAR(20),  -- sent, delivered, read
  timestamp TIMESTAMP,
  PRIMARY KEY (conversation_id, message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);
```

High-Level Diagram

```
┌──────────────┐
│   Client A   │
│ (E2E Crypto) │
└──────────────┘
       │ Persistent WebSocket
       ▼
┌──────────────────┐
│  Gateway Server  │ (Connection Manager)
│  (Load Balanced) │
└──────────────────┘
         │
         ▼
┌──────────────────┐
│  Message Router  │ (User ID → Gateway mapping)
└──────────────────┘
         │
    ┌────┴───────────────┬──────────────┐
    │                    │              │
    ▼                    ▼              ▼
┌─────────┐       ┌──────────┐    ┌──────────┐
│ Online? │       │ Offline  │    │  Group   │
│  Yes    │       │ Storage  │    │ Fanout   │
└─────────┘       └──────────┘    └──────────┘
     │                 │               │
     ▼                 ▼               │
┌─────────┐       ┌──────────┐        │
│ Direct  │       │  Kafka   │        │
│ Deliver │       │  Queue   │        │
└─────────┘       └──────────┘         │
     │                 │               │
     └─────────────────┴───────────────┘
                       ▼
               ┌──────────────┐
               │  Cassandra   │
               │  (Messages)  │
               └──────────────┘

Message Flow (1-to-1):
1. User A → Encrypt message with B's public key
2. Client A → Gateway A (WebSocket)
3. Gateway A → Message Router
4. Message Router:
   - Lookup B's gateway connection
   - If online: Forward to Gateway B → Client B
   - If offline: Write to Kafka → Storage
5. Store message in Cassandra (async)
6. Send delivery receipt to A
7. When B comes online:
   - Fetch pending messages from Kafka/Cassandra
   - Deliver via WebSocket
   - Send read receipt to A

Group Message Flow:
1. User A sends to Group G (50 members)
2. Message Router → Group Service: Get members
3. Group Fanout:
   - For each member: Route as 1-to-1 message
   - Async writes to Cassandra
   - If 50 members, creates 50 message copies
4. Optimization: Use message references
   - Store message once
   - 50 pointers to single message

Online Status (Redis):
SETEX user:123:online 60 "1"  # TTL 60 seconds
Client sends heartbeat every 30s to refresh

Heartbeat → If no heartbeat for 60s → Status = offline
Last seen = Last heartbeat timestamp
```

WebSocket Connection Management:

```java
import org.springframework.web.socket.*;
import org.springframework.data.redis.core.RedisTemplate;
import java.util.concurrent.*;
import java.util.*;

/**
 * WebSocket gateway server for real-time messaging.
 * Manages persistent connections and routes messages between users.
 */
@Component
public class GatewayServer implements WebSocketHandler {

    private final ConcurrentMap<String, WebSocketSession> connections =
        new ConcurrentHashMap<>();
    private final RedisTemplate<String, String> redisTemplate;
    private final MessageStorageService messageStorage;
    private final KafkaTemplate<String, Message> kafkaTemplate;

    private static final String GATEWAY_ID = System.getenv("GATEWAY_ID");
    private static final Duration ONLINE_STATUS_TTL =
Duration.ofSeconds(60);

    @Autowired
    public GatewayServer(RedisTemplate<String, String> redisTemplate,
                         MessageStorageService messageStorage,
                         KafkaTemplate<String, Message> kafkaTemplate) {
        this.redisTemplate = redisTemplate;
        this.messageStorage = messageStorage;
        this.kafkaTemplate = kafkaTemplate;
    }

    /**
     * Handle new WebSocket connection.
     * Registers user presence and delivers pending messages.
     */
    @Override
    public void afterConnectionEstablished(WebSocketSession session)
throws Exception {
        String userId = extractUserId(session);

        // Store connection locally
        connections.put(userId, session);

        // Register in Redis for routing from other gateways
        redisTemplate.opsForHash().put("user:gateway", userId,
GATEWAY_ID);

        // Set online status with TTL
        redisTemplate.opsForValue().set(
            "user:" + userId + ":online",
            "1",
            ONLINE_STATUS_TTL);

        // Deliver any pending messages
        List<Message> pendingMessages =
messageStorage.getPendingMessages(userId);
        for (Message message : pendingMessages) {
            sendMessage(session, message);
        }
    }

    /**
     * Handle incoming message from a connected user.
     * Routes to recipient if online, otherwise queues for later delivery.
     */
    @Override
    public void handleMessage(WebSocketSession session, WebSocketMessage<?
> webSocketMessage)
            throws Exception {
        String senderId = extractUserId(session);
        Message message =
parseMessage(webSocketMessage.getPayload().toString());
        message.setSenderId(senderId);
        message.setTimestamp(Instant.now());

        String recipientId = message.getRecipientId();

        // Find recipient's gateway
        String recipientGateway = (String) redisTemplate.opsForHash()
            .get("user:gateway", recipientId);

        if (recipientGateway != null) {
            // Recipient is online
            if (GATEWAY_ID.equals(recipientGateway)) {
                // Same gateway - deliver directly
                WebSocketSession recipientSession =
connections.get(recipientId);
                if (recipientSession != null && recipientSession.isOpen())
{
                    sendMessage(recipientSession, message);
                }
            } else {
                // Different gateway - route via inter-gateway messaging
                sendToGateway(recipientGateway, message);
            }
        } else {
            // Recipient offline - queue for later delivery
            kafkaTemplate.send("offline_messages", recipientId, message);
        }

        // Store in Cassandra asynchronously for persistence
        messageStorage.storeMessage(message);
    }

    /**
     * Handle connection close.
     * Cleans up presence data and updates last seen timestamp.
     */
    @Override
    public void afterConnectionClosed(WebSocketSession session,
CloseStatus status)
            throws Exception {
        String userId = extractUserId(session);

        connections.remove(userId);
        redisTemplate.opsForHash().delete("user:gateway", userId);
        redisTemplate.delete("user:" + userId + ":online");

        // Update last seen timestamp
        redisTemplate.opsForValue().set(
            "user:" + userId + ":last_seen",
            Instant.now().toString());
    }

    /**
     * Send message to inter-gateway messaging system for routing
     * to another gateway server.
     */
    private void sendToGateway(String gatewayId, Message message) {
        kafkaTemplate.send("gateway-messages-" + gatewayId, message);
    }

    /**
     * Listen for messages from other gateways destined for local users.
     */
    @KafkaListener(topics = "gateway-messages-${GATEWAY_ID}")
    public void handleInterGatewayMessage(Message message) {
        String recipientId = message.getRecipientId();
        WebSocketSession session = connections.get(recipientId);

        if (session != null && session.isOpen()) {
            try {
                sendMessage(session, message);
            } catch (IOException e) {
                // Log error and queue for retry
            }
        }
    }

    private void sendMessage(WebSocketSession session, Message message)
            throws IOException {
        String json = new ObjectMapper().writeValueAsString(message);
        session.sendMessage(new TextMessage(json));
    }

    private String extractUserId(WebSocketSession session) {
        return session.getAttributes().get("userId").toString();
    }

    private Message parseMessage(String payload) throws
JsonProcessingException {
        return new ObjectMapper().readValue(payload, Message.class);
    }

    @Override
    public void handleTransportError(WebSocketSession session, Throwable
exception) {
        // Log transport errors
    }

    @Override
    public boolean supportsPartialMessages() {
        return false;
    }
}

/**
 * Message entity for real-time messaging.
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Message {
    private String messageId;
    private String senderId;
    private String recipientId;
    private byte[] content; // Encrypted content
    private String mediaUrl;
    private MessageStatus status;
    private Instant timestamp;
}
```

```
public enum MessageStatus {
    SENT, DELIVERED, READ
}
```

End-to-End Encryption:

```
Key Exchange (Signal Protocol):
1. Each user generates:
   - Identity Key Pair (long-term)
   - Signed Pre-Key (medium-term)
   - One-Time Pre-Keys (ephemeral)
2. Keys uploaded to server
3. When A messages B:
   - Fetch B's public keys
   - Perform X3DH key agreement
   - Derive shared secret
   - Encrypt message with Double Ratchet
4. Server never sees plaintext

Message Encryption:
plaintext → AES-256-GCM → ciphertext
Server stores: ciphertext + metadata (sender, recipient, timestamp)
Only recipient's private key can decrypt
```

Trade-offs & Assumptions

- **WebSocket vs HTTP**: WebSocket for persistent connections; more efficient for messaging
- **Message Retention**: 30 days on server, then deleted; client stores locally
- **Group Size Limit**: 256 members; prevents fanout explosion
- **Media Compression**: Client-side compression before upload; reduces bandwidth
- **Assumption**: 70% messages delivered immediately (online users); 30% queued
- **Read Receipts**: Optional to preserve privacy; many users disable

---

14. Doctor Appointment Booking

Problem Overview

Design a system for booking doctor appointments with real-time availability, appointment reminders, patient history, and conflict prevention.

Back-of-the-Envelope Estimation

- **Doctors**: 100K doctors
- **Patients**: 10M registered
- **Appointments/day**: 500K bookings
- **Peak hours**: 9AM-11AM, 2PM-4PM

- **Avg appointment duration**: 30 minutes
- **Doctor availability**: 8 hours/day, 16 slots
- **Cancellation rate**: 15%

Functional Requirements

- **FR1**: View doctor availability by specialty, location, date
- **FR2**: Book appointments with conflict prevention
- **FR3**: Send appointment reminders (email, SMS, push)
- **FR4**: View patient history for doctors
- **FR5**: Handle cancellations and rescheduling
- **FR6**: Waitlist management for cancelled slots

Non-Functional Requirements

- **Scalability**: Handle 100K doctors, 10M patients
- **Availability**: 99.9% uptime
- **Latency**: <500ms for availability check
- **Consistency**: Strong consistency for bookings (no double bookings)
- **Reliability**: Guaranteed reminder delivery

High-Level Architecture

Components:

- **Client**: Web/Mobile apps
- **API Gateway**: Rate limiting, authentication
- **Doctor Service**: Doctor profiles, specialties
- **Appointment Service**: Booking management
- **Availability Service**: Real-time slot management
- **Notification Service**: Email/SMS/Push reminders
- **Patient Service**: Medical history, records
- **Payment Service**: Booking fees
- **Database**: PostgreSQL (core data), Redis (availability cache)

Data Storage Choices

Data Type	Storage	Justification
Appointments	PostgreSQL	ACID, complex queries, strong consistency
Doctor Availability	Redis + PostgreSQL	Fast reads, sync to DB
Patient Records	PostgreSQL + S3	Structured data + documents
Notification Queue	RabbitMQ	Reliable message delivery
Analytics	ClickHouse	Reporting, aggregations

Schema:

```
doctors (
  id UUID PRIMARY KEY,
  name VARCHAR(255),
  specialty VARCHAR(100),
  location VARCHAR(255),
  consultation_fee DECIMAL(10,2),
  years_experience INT
)

doctor_schedules (
  id UUID PRIMARY KEY,
  doctor_id UUID,
  day_of_week INT, -- 0-6
  start_time TIME,
  end_time TIME,
  slot_duration INT, -- minutes
  max_patients_per_slot INT
)

appointments (
  id UUID PRIMARY KEY,
  doctor_id UUID,
  patient_id UUID,
  appointment_date DATE,
  start_time TIME,
  end_time TIME,
  status VARCHAR(20), -- booked, confirmed, cancelled, completed
  notes TEXT,
  created_at TIMESTAMP,
  UNIQUE(doctor_id, appointment_date, start_time)
)

patients (
  id UUID PRIMARY KEY,
  name VARCHAR(255),
  email VARCHAR(255),
  phone VARCHAR(20),
  date_of_birth DATE,
  medical_history_url VARCHAR(500)
)

CREATE INDEX idx_appointments_doctor_date
ON appointments(doctor_id, appointment_date)
WHERE status IN ('booked', 'confirmed');
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
```

```
     │
     ▼
┌─────────────┐
│ API Gateway │
└──────┬──────┘
       │
   ┌───┴──────────┬──────────────┬──────────────┐
   ▼              ▼              ▼              ▼
┌────────┐  ┌──────────┐  ┌──────────┐  ┌──────────┐
│Doctor  │  │Appointmnt│  │  Patient │  │  Notif   │
│Service │  │ Service  │  │ Service  │  │ Service  │
└────┬───┘  └─────┬────┘  └─────┬────┘  └─────┬────┘
     │            │             │             │
     └────────────┴─────────────┴─────────────┘
                  │
                  ▼
          ┌─────────────┐
          │ PostgreSQL  │
          │ (ACID Txns) │
          └─────────────┘

Booking Flow (Optimistic Locking):
1. User searches: "Cardiologist in NYC, Dec 15"
2. Availability Service:
   - Query doctor_schedules
   - Check appointments table for conflicts
   - Return available slots
3. User selects slot: 10:00 AM
4. Appointment Service:
   BEGIN TRANSACTION
     INSERT INTO appointments
     (doctor_id, patient_id, date, start_time, status)
     VALUES (?, ?, ?, ?, 'booked')
     ON CONFLICT (doctor_id, date, start_time)
     DO NOTHING
     RETURNING id
   COMMIT
5. If id returned → Success
   If null → Slot already booked → Retry
6. Send confirmation email
7. Schedule reminder (24h before)

Availability Calculation:
def get_available_slots(doctor_id, date):
    # 1. Get doctor's schedule for day_of_week
    schedule = get_doctor_schedule(doctor_id, date.weekday())

    # 2. Generate all possible slots
    slots = []
    current = schedule.start_time
    while current < schedule.end_time:
        slots.append(current)
        current += timedelta(minutes=schedule.slot_duration)
    # 3. Query existing appointments
    booked = get_booked_appointments(doctor_id, date)

    # 4. Remove booked slots
    available = [s for s in slots if s not in booked]

    return available

Reminder System:
┌─────────────────┐
│ Cron Job        │
│ (Every hour)    │
└────────┬────────┘
         │
         ▼
  Query appointments WHERE
  appointment_date = CURRENT_DATE + 1
  AND status = 'booked'
  AND reminder_sent = FALSE
         │
         ▼
┌─────────────────┐
│  RabbitMQ       │
│ (Notification   │
│  Queue)         │
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│  Workers        │
│  - Email        │
│  - SMS          │
│  - Push Notif   │
└─────────────────┘
```

Cancellation and Waitlist:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.time.*;
import java.util.*;

/**
 * Service handling appointment cancellations with automatic waitlist
management.
 * Notifies waitlisted patients when slots become available.
 */
@Service
public class AppointmentCancellationService {

    private final AppointmentRepository appointmentRepository;
    private final WaitlistRepository waitlistRepository;
    private final NotificationService notificationService;
    private final BookingService bookingService;

    @Autowired
    public AppointmentCancellationService(AppointmentRepository
appointmentRepository,
                                          WaitlistRepository
waitlistRepository,
                                          NotificationService
notificationService,
                                          BookingService bookingService) {
        this.appointmentRepository = appointmentRepository;
        this.waitlistRepository = waitlistRepository;
        this.notificationService = notificationService;
        this.bookingService = bookingService;
    }

    /**
     * Cancel an appointment and offer the freed slot to waitlisted
patients.
     * Supports auto-booking for patients who opted in.
     */
    @Transactional
    public CancellationResult cancelAppointment(UUID appointmentId, String
cancelledBy) {
        // Step 1: Update appointment status
        Appointment appointment =
appointmentRepository.findById(appointmentId)
            .orElseThrow(() -> new
AppointmentNotFoundException(appointmentId));

        appointment.setStatus(AppointmentStatus.CANCELLED);
        appointment.setCancelledAt(Instant.now());
        appointment.setCancelledBy(cancelledBy);
        appointmentRepository.save(appointment);

        // Step 2: Check waitlist for this slot
        Optional<WaitlistEntry> waitlistEntry = waitlistRepository
            .findFirstByDoctorIdAndPreferredDateOrderByCreatedAtAsc(
                appointment.getDoctorId(),
                appointment.getAppointmentDate());

        if (waitlistEntry.isPresent()) {
            WaitlistEntry entry = waitlistEntry.get();

            // Step 3: Notify waitlisted patient about availability
            String message = String.format(
                "A slot is now available on %s at %s",
                appointment.getAppointmentDate(),
                appointment.getStartTime());

            notificationService.sendNotification(
                entry.getPatientId(),
                NotificationType.SLOT_AVAILABLE,
                message);

            // Step 4: Auto-book if patient configured this preference
            if (entry.isAutoBook()) {
                try {
                    Booking newBooking = bookingService.createBooking(
                        entry.getPatientId(),
                        appointment.getDoctorId(),
                        appointment.getAppointmentDate(),
                        appointment.getStartTime());

                    // Remove from waitlist after successful booking
                    waitlistRepository.delete(entry);

                    return CancellationResult.withAutoBooking(appointment, 
newBooking);
                } catch (BookingException e) {
                    // Auto-booking failed, just notify
                    return 
CancellationResult.withNotification(appointment, entry.getPatientId());
                }
            }

            return CancellationResult.withNotification(appointment, 
entry.getPatientId());
        }

        return CancellationResult.cancelled(appointment);
    }

    /**
     * Add patient to waitlist for a specific doctor and date.
     */
    @Transactional
    public WaitlistEntry addToWaitlist(UUID patientId, UUID doctorId,
                                        LocalDate preferredDate, boolean 
autoBook) {
        // Check if already on waitlist
        if 
(waitlistRepository.existsByPatientIdAndDoctorIdAndPreferredDate(
                patientId, doctorId, preferredDate)) {
            throw new DuplicateWaitlistEntryException();
        }

        WaitlistEntry entry = WaitlistEntry.builder()
            .patientId(patientId)
            .doctorId(doctorId)
            .preferredDate(preferredDate)
            .autoBook(autoBook)
            .createdAt(Instant.now())
            .build();

        return waitlistRepository.save(entry);
    }
```

```
}

/**
 * Result of a cancellation operation, including any auto-booking actions.
 */
@Data
@Builder
public class CancellationResult {
    private Appointment cancelledAppointment;
    private Booking autoBookedAppointment;
    private UUID notifiedPatientId;
    private CancellationOutcome outcome;

    public static CancellationResult cancelled(Appointment appointment) {
        return CancellationResult.builder()
            .cancelledAppointment(appointment)
            .outcome(CancellationOutcome.CANCELLED)
            .build();
    }

    public static CancellationResult withNotification(Appointment
appointment,
                                                       UUID patientId) {
        return CancellationResult.builder()
            .cancelledAppointment(appointment)
            .notifiedPatientId(patientId)
            .outcome(CancellationOutcome.WAITLIST_NOTIFIED)
            .build();
    }

    public static CancellationResult withAutoBooking(Appointment
appointment,
                                                      Booking booking) {
        return CancellationResult.builder()
            .cancelledAppointment(appointment)
            .autoBookedAppointment(booking)
            .outcome(CancellationOutcome.AUTO_BOOKED)
            .build();
    }
}

public enum CancellationOutcome {
    CANCELLED, WAITLIST_NOTIFIED, AUTO_BOOKED
}

/**
 * Entity representing a waitlist entry for appointment slots.
 */
@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class WaitlistEntry {
    @Id
    @GeneratedValue
    private UUID id;

    private UUID patientId;
    private UUID doctorId;
    private LocalDate preferredDate;
    private boolean autoBook;
    private Instant createdAt;
}
```

Trade-offs & Assumptions

- **Unique Constraint**: Database-level prevents double bookings; race conditions handled by DB
- **Availability Cache**: Redis cache for popular doctors; 5 min TTL
- **Reminder Timing**: 24h before + 1h before; configurable per patient
- **No-show Handling**: Automatic status update; track no-show rate per patient
- **Assumption**: 85% appointments are booked 1-7 days in advance; optimize for this window

---

15. Hotel Reservation System

Problem Overview

Design a hotel reservation system that prevents double bookings through robust locking mechanisms, handles concurrent booking requests, and manages room inventory across multiple properties.

Back-of-the-Envelope Estimation

- **Hotels**: 50K properties
- **Rooms**: 10M total rooms
- **Bookings/day**: 500K reservations
- **Peak bookings/sec**: 500K / 86400 × 10 = ~60 bookings/sec
- **Concurrent requests**: 1000 users trying to book same room
- **Average stay**: 3 nights

Functional Requirements

- **FR1**: Search available rooms by location, dates, guests
- **FR2**: Book rooms with guarantee of no double booking
- **FR3**: Hold rooms temporarily during booking process
- **FR4**: Handle cancellations and modifications
- **FR5**: Manage overbooking policies

Non-Functional Requirements

- **Scalability**: Handle 500K bookings/day
- **Availability**: 99.95% uptime
- **Latency**: <1s for booking confirmation

- **Consistency**: Strong consistency for inventory (no double bookings)
- **Isolation**: Prevent race conditions under high concurrency

High-Level Architecture

Components:

- **Client**: Web/Mobile booking interface
- **API Gateway**: Load balancing, rate limiting
- **Search Service**: Room availability queries
- **Booking Service**: Reservation management
- **Inventory Service**: Room availability tracking
- **Lock Service**: Distributed locking (Redis)
- **Payment Service**: Payment processing
- **Database**: PostgreSQL (ACID transactions)

Data Storage Choices

Data Type	Storage	Justification
Room Inventory	PostgreSQL	Strong consistency, ACID
Booking Locks	Redis	Fast distributed locking, TTL
Reservations	PostgreSQL	Transactional integrity
Search Cache	Elasticsearch	Fast availability queries

Schema:

```
hotels (
  id BIGINT PRIMARY KEY,
  name VARCHAR(255),
  location VARCHAR(255),
  star_rating INT
)

rooms (
  id BIGINT PRIMARY KEY,
  hotel_id BIGINT,
  room_number VARCHAR(20),
  room_type VARCHAR(50),
  max_occupancy INT,
  base_price DECIMAL(10,2)
)

room_inventory (
  room_id BIGINT,
  date DATE,
  total_rooms INT,
  available_rooms INT,
  PRIMARY KEY (room_id, date)
)

reservations (
  id UUID PRIMARY KEY,
  hotel_id BIGINT,
  room_id BIGINT,
  user_id UUID,
  check_in DATE,
  check_out DATE,
  num_rooms INT,
  status VARCHAR(20), -- pending, confirmed, cancelled
  total_price DECIMAL(10,2),
  created_at TIMESTAMP
)

CREATE INDEX idx_inventory_availability
ON room_inventory(room_id, date)
WHERE available_rooms > 0;
```

High-Level Diagram

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌─────────────┐
│ API Gateway │
└──────┬──────┘
       │
   ┌───┴──────────┬──────────────┐
   ▼              ▼              ▼
┌────────┐  ┌──────────┐  ┌──────────┐
│ Search │  │ Booking  │  │Inventory │
│Service │  │ Service  │  │ Service  │
└───┬────┘  └────┬─────┘  └────┬─────┘
    │            │             │
    │       ┌────▼────┐        │
    │       │  Redis  │        │
    │       │ (Lock)  │        │
    │       └─────────┘        │
    │                          │
    └──────────┬───────────────┘
               ▼
      ┌──────────────┐
      │  PostgreSQL  │
      │(Serializable │
      │  Isolation)  │
      └──────────────┘

Double Booking Prevention (Multi-Layer):
┌──────────────────────────────────────┐
│ Layer 1: Distributed Lock (Redis)    │
│ - Acquire before booking attempt     │
│ - TTL: 30 seconds                    │
└──────────────────────────────────────┘

┌──────────────────────────────────────┐
│ Layer 2: DB-level Constraint         │
│ - UNIQUE (room_id, date)             │
│ - CHECK (available_rooms >= 0)       │
└──────────────────────────────────────┘

┌──────────────────────────────────────┐
│ Layer 3: Serializable Isolation      │
│ - BEGIN TRANSACTION ISOLATION LEVEL  │
│   SERIALIZABLE                       │
└──────────────────────────────────────┘

┌──────────────────────────────────────┐
│ Layer 4: Optimistic Locking          │
│ - Version field on inventory         │
│ - UPDATE WHERE version = old_version│
└──────────────────────────────────────┘

Booking Flow:
1. User selects: Room 101, Dec 15-17 (2 nights)
2. Acquire distributed lock:
   lock_key = "room:101:2024-12-15:2024-12-17"
   acquired = SETNX lock_key user_session_id EX 30
3. If lock acquired:
   BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE
     -- Check availability
     SELECT available_rooms
     FROM room_inventory
     WHERE room_id = 101
       AND date BETWEEN '2024-12-15' AND '2024-12-16'
     FOR UPDATE

     -- Verify all dates have availability
     IF all dates have available_rooms > 0:
       -- Decrement inventory for each night
       UPDATE room_inventory
       SET available_rooms = available_rooms - 1
       WHERE room_id = 101
         AND date BETWEEN '2024-12-15' AND '2024-12-16'

       -- Create reservation
       INSERT INTO reservations (...)

       -- Process payment
       payment_result = process_payment(...)

       IF payment_successful:
         COMMIT
         release_lock(lock_key)
         return SUCCESS
       ELSE:
         ROLLBACK
         release_lock(lock_key)
         return PAYMENT_FAILED
     ELSE:
       ROLLBACK
       release_lock(lock_key)
       return NO_AVAILABILITY
   END TRANSACTION
4. If lock not acquired:
   WAIT 100ms, RETRY (max 3 attempts)
   return BOOKING_IN_PROGRESS
```

Distributed Lock Implementation:

```java
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.stereotype.Component;
import java.time.*;
import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 * Distributed lock implementation for hotel room reservations.
 * Uses Redis with Lua scripts for atomic operations.
 */
@Component
public class HotelDistributedLock {

    private final RedisTemplate<String, String> redisTemplate;

    /**
     * Lua script for atomic release - only releases if caller owns the
lock.
     */
    private static final String RELEASE_SCRIPT = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("DEL", KEYS[1])
        else
            return 0
        end
        """;

    /**
     * Lua script for atomic extend - only extends if caller owns the
lock.
     */
    private static final String EXTEND_SCRIPT = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("EXPIRE", KEYS[1], ARGV[2])
        else
            return 0
        end
        """;

    public HotelDistributedLock(RedisTemplate<String, String>
redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    /**
     * Acquire a lock for a room booking with automatic expiry.
     *
     * @param lockKey Unique key for the lock (e.g., "room:101:2024-12-
15:2024-12-17")
     * @param value Unique identifier for the lock holder (session ID)
     * @param ttlSeconds Lock expiry time in seconds
     * @return true if lock acquired successfully
     */
    public boolean acquire(String lockKey, String value, int ttlSeconds) {
        Boolean result = redisTemplate.opsForValue()
            .setIfAbsent(lockKey, value, ttlSeconds, TimeUnit.SECONDS);
        return Boolean.TRUE.equals(result);
    }

    /**
     * Release a lock only if the caller owns it.
     * Prevents accidental release of locks held by other processes.
     */
    public void release(String lockKey, String value) {
        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(RELEASE_SCRIPT, Long.class);
        redisTemplate.execute(script, Collections.singletonList(lockKey),
value);
    }

    /**
     * Extend lock TTL if still owned by the caller.
     * Useful for long-running booking operations.
     */
    public boolean extend(String lockKey, String value, int ttlSeconds) {
        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(EXTEND_SCRIPT, Long.class);
        Long result = redisTemplate.execute(
            script,
            Collections.singletonList(lockKey),
            value,
            String.valueOf(ttlSeconds));
        return result != null && result == 1;
    }
}
```

Optimistic Locking with Version:

```sql
-- Alternative approach using version field
ALTER TABLE room_inventory ADD COLUMN version INT DEFAULT 1;

-- Booking attempt
UPDATE room_inventory
SET available_rooms = available_rooms - 1,
    version = version + 1
WHERE room_id = ?
  AND date = ?
  AND version = ?  -- Old version
  AND available_rooms > 0;

-- If affected_rows = 0, concurrent modification detected
-- Retry with fresh version
```

Trade-offs & Assumptions

- **Pessimistic Lock (Redis)**: Prevents concurrent attempts; 30s TTL prevents deadlocks
- **Serializable Isolation**: Strongest guarantee but performance cost; use only for bookings
- **Lock Granularity**: Lock entire date range, not individual dates; simpler but coarser
- **Overbooking**: Intentional 5-10% overbooking to handle cancellations; needs careful tuning
- **Assumption**: 95% of bookings complete within 30 seconds; lock TTL sufficient

---

---

16. Local vs Global Caching

Concept Overview

Local caching stores data on individual application servers, while global caching uses a centralized cache shared across all servers. Understanding when to use each is critical for system performance.

Local Caching

Characteristics:

- **Location**: In-process memory (e.g., HashMap, LRU cache)
- **Access Time**: Sub-microsecond (50-100 nanoseconds)
- **Scope**: Single application instance
- **Consistency**: No coordination needed
- **Capacity**: Limited by server RAM (typically 1-10GB)

Use Cases:

- Application configuration
- Frequently accessed reference data (rarely changes)
- User session data (sticky sessions)
- Computed results (memoization)

Pros:

- Extremely fast (no network)
- No single point of failure
- Free (uses existing memory)
- Zero latency

Cons:

- Data duplication across servers
- Cache invalidation challenges
- Limited capacity per server
- Inconsistency across instances

Global Caching

Characteristics:

- **Location**: Centralized service (Redis, Memcached)
- **Access Time**: 1-5 milliseconds (network hop)
- **Scope**: Shared across all application servers
- **Consistency**: Single source of truth
- **Capacity**: Virtually unlimited (cluster horizontally)

Use Cases:

- User sessions (any server can handle request)
- Rate limiting counters
- Real-time data (stock prices, inventory)
- Shared state across microservices

Pros:

- Consistent data across all servers
- Better cache hit rate (pooled requests)
- Easier cache invalidation
- Scales independently

Cons:

- Network latency (1-5ms)
- Single point of failure (mitigate with clustering)
- Additional infrastructure cost
- Network bandwidth usage

Hybrid Approach (Multi-Level Caching)

Common Pattern:

```
Request Flow:
1. Check L1 (Local Cache - in-memory)
   └─ Hit: Return immediately (0.1ms)
```

```
    └─ Miss: Check L2
2. Check L2 (Global Cache - Redis)
    └─ Hit: Store in L1, return (2ms)
    └─ Miss: Check L3
3. Check L3 (Database)
    └─ Query DB, store in L2 and L1, return (50ms)
```

Example Implementation:

```java
import com.github.benmanes.caffeine.cache.Cache;
import com.github.benmanes.caffeine.cache.Caffeine;
import org.springframework.data.redis.core.RedisTemplate;
import java.time.Duration;
import java.util.Optional;
import java.util.concurrent.TimeUnit;
import java.util.function.Supplier;

/**
 * Multi-level caching implementation with L1 (local), L2 (Redis), and L3 (database).
 * Provides optimal performance by checking fastest storage first.
 *
 * Access Time Comparison:
 * - L1 (Local): ~0.1ms
 * - L2 (Redis): ~2ms
 * - L3 (Database): ~50ms
 */
public class MultiLevelCache<K, V> {

    // L1: In-process local cache (Caffeine)
    private final Cache<K, V> localCache;

    // L2: Distributed cache (Redis)
    private final RedisTemplate<String, V> redisTemplate;
    private final String cachePrefix;

    // L3: Database (accessed via supplier function)
    private final Duration l2CacheTtl;

    public MultiLevelCache(RedisTemplate<String, V> redisTemplate,
                           String cachePrefix,
                           int localCacheCapacity,
                           Duration localCacheTtl,
                           Duration l2CacheTtl) {
        this.redisTemplate = redisTemplate;
        this.cachePrefix = cachePrefix;
        this.l2CacheTtl = l2CacheTtl;

        // Configure L1 cache with size limit and expiry
        this.localCache = Caffeine.newBuilder()
            .maximumSize(localCacheCapacity)
            .expireAfterWrite(localCacheTtl)
```

```
                .recordStats()
                .build();
    }

    /**
     * Get value from cache hierarchy, loading from database if necessary.
     *
     * @param key Cache key
     * @param databaseLoader Function to load from database on cache miss
     * @return Optional containing the value if found/loaded
     */
    public Optional<V> get(K key, Supplier<Optional<V>> databaseLoader) {
        // L1: Check local cache first (fastest)
        V value = localCache.getIfPresent(key);
        if (value != null) {
            return Optional.of(value);
        }

        // L2: Check Redis (distributed cache)
        String redisKey = buildRedisKey(key);
        value = redisTemplate.opsForValue().get(redisKey);
        if (value != null) {
            // Populate L1 for subsequent requests
            localCache.put(key, value);
            return Optional.of(value);
        }

        // L3: Load from database
        Optional<V> dbResult = databaseLoader.get();
        if (dbResult.isPresent()) {
            value = dbResult.get();
            // Populate both L1 and L2 caches
            localCache.put(key, value);
            redisTemplate.opsForValue().set(redisKey, value, l2CacheTtl);
        }

        return dbResult;
    }

    /**
     * Put value directly into all cache levels.
     * Use after database writes to keep caches warm.
     */
    public void put(K key, V value) {
        localCache.put(key, value);
        String redisKey = buildRedisKey(key);
        redisTemplate.opsForValue().set(redisKey, value, l2CacheTtl);
    }

    /**
     * Invalidate a key across all cache levels.
     * Call this when the underlying data changes.
     */
    public void invalidate(K key) {
        localCache.invalidate(key);
        String redisKey = buildRedisKey(key);
        redisTemplate.delete(redisKey);
    }

    /**
     * Invalidate all entries in L1 cache.
     * L2 entries will expire via TTL.
     */
    public void invalidateAll() {
        localCache.invalidateAll();
    }

    private String buildRedisKey(K key) {
        return cachePrefix + ":" + key.toString();
    }

    /**
     * Get cache statistics for monitoring.
     */
    public CacheStatistics getStatistics() {
        var stats = localCache.stats();
        return new CacheStatistics(
            stats.hitRate(),
            stats.missRate(),
            stats.evictionCount(),
            localCache.estimatedSize()
        );
    }
}

/**
 * Cache statistics for monitoring dashboard.
 */
public record CacheStatistics(
    double hitRate,
    double missRate,
    long evictionCount,
    long currentSize
) {}

/**
 * Example usage with a User entity.
 */
@Service
public class UserService {

    private final MultiLevelCache<UUID, User> userCache;
    private final UserRepository userRepository;

    @Autowired
    public UserService(RedisTemplate<String, User> redisTemplate,
                      UserRepository userRepository) {
        this.userRepository = userRepository;
        this.userCache = new MultiLevelCache<>(
            redisTemplate,
            "user",            // Redis key prefix
            1000,              // L1 capacity
            Duration.ofMinutes(5),  // L1 TTL
            Duration.ofHours(1)     // L2 TTL
        );
    }

    public Optional<User> getUser(UUID userId) {
        return userCache.get(userId,
            () -> userRepository.findById(userId));
    }

    @Transactional
    public User updateUser(UUID userId, UserUpdateRequest request) {
        User user = userRepository.findById(userId)
            .orElseThrow(() -> new UserNotFoundException(userId));

        user.setName(request.getName());
        user.setEmail(request.getEmail());
        user = userRepository.save(user);

        // Update cache after write
        userCache.put(userId, user);

        return user;
    }

    @Transactional
    public void deleteUser(UUID userId) {
        userRepository.deleteById(userId);
        userCache.invalidate(userId);
    }
}
```

Comparison Table

Aspect	Local Cache	Global Cache	Multi-Level
Latency	0.0001ms	1-5ms	0.0001-5ms
Consistency	Poor	Good	Medium
Scalability	Limited	Excellent	Good
Fault Tolerance	High	Medium	High
Cost	Free	$	$$
Complexity	Low	Medium	High
Hit Rate	Lower	Higher	Highest

Cache Invalidation Strategies

Local Cache Invalidation:

```
1. TTL-based: Expire after N seconds
2. Event-driven: Pub/Sub notifications
3. Version-based: Increment version on update
4. Manual: Clear cache on write
```

Global Cache Invalidation:

```
1. TTL: Redis EXPIRE command
2. Write-through: Update cache on DB write
3. Write-behind: Async cache update
4. Cache-aside: Invalidate on write, lazy load on read
```

Trade-offs & Recommendations

Use Local Cache When:

- Data is read-heavy and rarely changes
- Latency is critical (microseconds matter)
- Data size is small
- Inconsistency is acceptable

Use Global Cache When:

- Data consistency is required
- Multiple services need same data
- Rate limiting or counters
- Session management without sticky routing

Use Multi-Level Cache When:

- Highest performance needed
- Can tolerate some inconsistency
- Traffic patterns have hot spots
- Budget allows complexity

---

17. Sharding and Federation

Sharding (Horizontal Partitioning)

Concept: Split a large database into smaller, independent pieces (shards) based on a shard key.

Sharding Strategies:

1. **Range-Based Sharding**:

```
User IDs 1-1M     → Shard 1
User IDs 1M-2M    → Shard 2
User IDs 2M-3M    → Shard 3
```

Pros: Simple, easy range queries **Cons**: Hotspots (new users always in last shard)

2. **Hash-Based Sharding**:

```
hash(user_id) % num_shards
user_123 → hash(123) % 4 = 3 → Shard 3
user_456 → hash(456) % 4 = 0 → Shard 0
```

Pros: Even distribution **Cons**: Range queries difficult, resharding painful

3. **Geographic Sharding**:

```
US users      → US Shard
EU users      → EU Shard
APAC users    → APAC Shard
```

Pros: Low latency, data locality **Cons**: Uneven distribution, cross-region queries expensive

4. **Consistent Hashing**:

```
Hash Ring:
Shard 1: positions 0-250
Shard 2: positions 251-500
Shard 3: positions 501-750
Shard 4: positions 751-999

user_id → hash(user_id) % 1000 → position → shard
```

Pros: Minimal data movement when resharding **Cons**: Implementation complexity

Sharding Implementation:

```java
import java.util.*;
import java.sql.*;
import javax.sql.DataSource;

/**
 * Router for hash-based database sharding.
 * Distributes data across multiple database shards using consistent
hashing.
 */
public class ShardRouter {

    private final List<DataSource> shards;
    private final int numShards;

    public ShardRouter(List<DataSource> shards) {
        this.shards = new ArrayList<>(shards);
        this.numShards = shards.size();
    }

    /**
     * Determine which shard contains data for a given user.
     * Uses hash-based routing for even distribution.
     */
    public DataSource getShardForUser(UUID userId) {
        int shardId = Math.abs(userId.hashCode() % numShards);
        return shards.get(shardId);
    }

    /**
     * Query a specific user from their designated shard.
     */
    public Optional<User> queryUser(UUID userId) throws SQLException {
        DataSource shard = getShardForUser(userId);

        String sql = "SELECT * FROM users WHERE id = ?";
        try (Connection conn = shard.getConnection();
             PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setObject(1, userId);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                return Optional.of(mapToUser(rs));
            }
            return Optional.empty();
        }
    }

    /**
     * Execute a query across all shards (scatter-gather pattern).
     * Use sparingly as this is expensive.
     */
    public List<User> queryAllShards(String sql, Object... params) throws
SQLException {
        List<User> results = new ArrayList<>();

        for (DataSource shard : shards) {
            try (Connection conn = shard.getConnection();
                 PreparedStatement stmt = conn.prepareStatement(sql)) {
```

```
                for (int i = 0; i < params.length; i++) {
                    stmt.setObject(i + 1, params[i]);
                }

                ResultSet rs = stmt.executeQuery();
                while (rs.next()) {
                    results.add(mapToUser(rs));
                }
            }
        }

        return results;
    }

    /**
     * Execute query across all shards in parallel for better performance.
     */
    public List<User> queryAllShardsParallel(String sql, Object... params)
{
        return shards.parallelStream()
            .flatMap(shard -> {
                try {
                    return queryOnShard(shard, sql, params).stream();
                } catch (SQLException e) {
                    throw new RuntimeException("Shard query failed", e);
                }
            })
            .collect(Collectors.toList());
    }

    private List<User> queryOnShard(DataSource shard, String sql, Object[]
params)
            throws SQLException {
        List<User> results = new ArrayList<>();

        try (Connection conn = shard.getConnection();
             PreparedStatement stmt = conn.prepareStatement(sql)) {

            for (int i = 0; i < params.length; i++) {
                stmt.setObject(i + 1, params[i]);
            }

            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                results.add(mapToUser(rs));
            }
        }

        return results;
    }

    private User mapToUser(ResultSet rs) throws SQLException {
        return new User(
            UUID.fromString(rs.getString("id")),
            rs.getString("name"),
            rs.getString("email")
        );
    }
}
```

Challenges:

- **Cross-shard queries**: Requires scatter-gather pattern
- **Transactions**: Difficult across shards; use Saga pattern
- **Rebalancing**: Adding/removing shards requires data migration
- **Schema changes**: Must coordinate across all shards

Federation (Functional Partitioning)

Concept: Split database by function/domain, not by data volume.

Example:

```
Database 1: User Service (users, auth, profiles)
Database 2: Order Service (orders, payments)
Database 3: Inventory Service (products, stock)
Database 4: Analytics Service (events, metrics)
```

Federation Implementation:

```
/**
 * Each microservice has its own dedicated database.
 * Services communicate via APIs, not direct DB joins.
 */
@Service
public class UserService {

    private final JdbcTemplate db;

    public UserService(@Qualifier("userDataSource") DataSource dataSource)
{
        this.db = new JdbcTemplate(dataSource);
    }

    public Optional<User> getUser(Long userId) {
        String sql = "SELECT * FROM users WHERE id = ?";
        List<User> users = db.query(sql, new UserRowMapper(), userId);
        return users.isEmpty() ? Optional.empty() :
Optional.of(users.get(0));
    }
}

@Service
```

```
public class OrderService {

    private final JdbcTemplate db;

    public OrderService(@Qualifier("orderDataSource") DataSource
dataSource) {
        this.db = new JdbcTemplate(dataSource);
    }

    public List<Order> getOrders(Long userId) {
        String sql = "SELECT * FROM orders WHERE user_id = ?";
        return db.query(sql, new OrderRowMapper(), userId);
    }
}
```

Pros:

- Clear separation of concerns
- Independent scaling per service
- Easier to understand and maintain
- Aligns with microservices

Cons:

- Cross-database joins impossible
- Data duplication needed
- Distributed transactions complex
- Need to maintain referential integrity manually

Comparison

Aspect	Sharding	Federation
Purpose	Scale single table/DB	Separate by domain
Data Split	Horizontal	Vertical
Queries	Within shard fast	Within service fast
Joins	Difficult	Impossible cross-DB
Complexity	High (data distribution)	Medium (service boundaries)
Use Case	Massive single table	Microservices

Availability Challenges

Sharding Availability:

- **Problem**: Shard failure = partial data loss
- **Solution**: Replicate each shard (master-slave)

```
Shard 1: Master + 2 Slaves
Shard 2: Master + 2 Slaves
Shard 3: Master + 2 Slaves
```

- **Trade-off**: 3x storage cost for high availability

Federation Availability:

- **Problem**: Service failure = feature unavailable
- **Solution**: Circuit breaker, graceful degradation

```java
// Graceful degradation with circuit breaker pattern
public List<Order> getOrdersWithFallback(UUID userId) {
    try {
        return orderService.getOrders(userId);
    } catch (ServiceUnavailableException e) {
        logger.error("Order service down", e);
        return Collections.emptyList();  // Graceful degradation
    }
}
```

When to Use Each

Use Sharding When:

- Single table > 100 million rows
- Query performance degrading
- Need to scale horizontally
- Data naturally partitions by key (user_id, tenant_id)

Use Federation When:

- Building microservices
- Clear domain boundaries
- Different scaling needs per service
- Want team autonomy

---

18. Caching Techniques

Caching Strategies

1. Cache-Aside (Lazy Loading)

Pattern:

```
/**
 * Cache-aside pattern implementation.
```

```java
 * Application manages cache population explicitly.
 */
@Service
public class UserCacheAsideService {

    private final RedisTemplate<String, User> cache;
    private final UserRepository userRepository;

    private static final Duration CACHE_TTL = Duration.ofHours(1);

    public UserCacheAsideService(RedisTemplate<String, User> cache,
                                 UserRepository userRepository) {
        this.cache = cache;
        this.userRepository = userRepository;
    }

    /**
     * Get user with cache-aside pattern.
     * Check cache first, load from DB on miss.
     */
    public Optional<User> getUser(UUID userId) {
        String cacheKey = "user:" + userId;

        // Try cache first
        User cachedUser = cache.opsForValue().get(cacheKey);
        if (cachedUser != null) {
            return Optional.of(cachedUser);
        }

        // Cache miss - query database
        Optional<User> userOpt = userRepository.findById(userId);

        // Populate cache for future requests
        userOpt.ifPresent(user ->
            cache.opsForValue().set(cacheKey, user, CACHE_TTL));

        return userOpt;
    }

    /**
     * Update user and invalidate cache.
     */
    @Transactional
    public User updateUser(UUID userId, UserUpdateRequest request) {
        // Update database
        User user = userRepository.findById(userId)
            .orElseThrow(() -> new UserNotFoundException(userId));

        user.setName(request.getName());
        user.setEmail(request.getEmail());
        user = userRepository.save(user);

        // Invalidate cache - next read will reload from DB
        String cacheKey = "user:" + userId;
        cache.delete(cacheKey);

        return user;
    }
}
```

Pros: Only caches requested data, cache resilience **Cons**: Cache miss penalty, stale data possible

2. Write-Through Cache

Pattern:

```
/**
 * Write-through cache pattern.
 * Cache is updated synchronously with database writes.
 */
@Service
public class UserWriteThroughService {

    private final RedisTemplate<String, User> cache;
    private final UserRepository userRepository;

    private static final Duration CACHE_TTL = Duration.ofHours(1);

    /**
     * Update user with write-through pattern.
     * Both cache and DB are updated in the same operation.
     */
    @Transactional
    public User updateUser(UUID userId, UserUpdateRequest request) {
        User user = userRepository.findById(userId)
            .orElseThrow(() -> new UserNotFoundException(userId));

        user.setName(request.getName());
        user.setEmail(request.getEmail());

        // Write to cache first
        String cacheKey = "user:" + userId;
        cache.opsForValue().set(cacheKey, user, CACHE_TTL);

        // Write to database (synchronously)
        user = userRepository.save(user);

        return user;
    }
}
```

Pros: Cache always consistent with DB **Cons**: Write latency (two writes), wasted cache space

3. Write-Behind (Write-Back) Cache

Pattern:

```
/**
 * Write-behind cache pattern.
 * Writes go to cache immediately, DB is updated asynchronously.
 */
@Service
public class UserWriteBehindService {

    private final RedisTemplate<String, User> cache;
    private final KafkaTemplate<String, UserWriteEvent> writeQueue;

    private static final Duration CACHE_TTL = Duration.ofHours(1);

    /**
     * Update user with write-behind pattern.
     * Fast response - DB write is queued for async processing.
     */
    public User updateUser(UUID userId, UserUpdateRequest request) {
        User user = new User(userId, request.getName(),
request.getEmail());

        // Write to cache immediately
        String cacheKey = "user:" + userId;
        cache.opsForValue().set(cacheKey, user, CACHE_TTL);

        // Queue DB write asynchronously
        UserWriteEvent event = new UserWriteEvent(
            "users",
            userId,
            user,
            Instant.now()
        );
        writeQueue.send("db-writes", userId.toString(), event);

        return user; // Fast response
    }
}

/**
 * Background worker processing queued DB writes.
 */
@Component
public class DatabaseWriteWorker {

    private final UserRepository userRepository;

    @KafkaListener(topics = "db-writes", groupId = "db-write-worker")
    public void processWrite(UserWriteEvent event) {
        userRepository.save(event.getUser());
    }
}
```

```
/**
 * Event for queued database writes.
 */
public record UserWriteEvent(
    String table,
    UUID id,
    User user,
    Instant timestamp
) {}
```

Pros: Fast writes, batching possible **Cons**: Data loss risk, complexity

4. Read-Through Cache

Pattern:

```
/**
 * Read-through cache using Spring's @Cacheable annotation.
 * Cache layer handles DB queries automatically.
 */
@Service
public class UserReadThroughService {

    private final UserRepository userRepository;

    /**
     * Get user with read-through pattern.
     * Spring Cache abstraction handles cache management.
     */
    @Cacheable(value = "users", key = "#userId", unless = "#result ==
null")
    public User getUser(UUID userId) {
        return userRepository.findById(userId)
            .orElse(null);
    }

    @CacheEvict(value = "users", key = "#userId")
    public void evictUser(UUID userId) {
        // Cache entry will be evicted
    }

    @CachePut(value = "users", key = "#user.id")
    public User updateUser(User user) {
        return userRepository.save(user);
    }
}
```

Pros: Simplified application code **Cons**: Tight coupling, less control

Cache Eviction Policies

1. LRU (Least Recently Used)

```java
import java.util.*;

/**
 * Thread-safe LRU Cache implementation using LinkedHashMap.
 * Evicts least recently accessed entries when capacity is reached.
 */
public class LRUCache<K, V> {

    private final int capacity;
    private final Map<K, V> cache;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        // LinkedHashMap with access-order (true) maintains LRU order
        this.cache = Collections.synchronizedMap(
            new LinkedHashMap<K, V>(capacity, 0.75f, true) {
                @Override
                protected boolean removeEldestEntry(Map.Entry<K, V>
eldest) {
                    return size() > LRUCache.this.capacity;
                }
            });
    }

    public V get(K key) {
        return cache.get(key); // Access updates recency in LinkedHashMap
    }

    public void put(K key, V value) {
        cache.put(key, value);
        // Eldest entry auto-removed if over capacity
    }

    public void remove(K key) {
        cache.remove(key);
    }

    public int size() {
        return cache.size();
    }

    public void clear() {
        cache.clear();
    }
}
```

2. LFU (Least Frequently Used)

```java
import java.util.*;

/**
 * LFU Cache implementation with O(1) get and put operations.
 * Evicts least frequently used entries; ties broken by recency.
 */
public class LFUCache<K, V> {

    private final int capacity;
    private int minFrequency;

    // Key -> (Value, Frequency)
    private final Map<K, CacheEntry<V>> cache;

    // Frequency -> LinkedHashSet of keys (preserves insertion order)
    private final Map<Integer, LinkedHashSet<K>> frequencyMap;

    public LFUCache(int capacity) {
        this.capacity = capacity;
        this.minFrequency = 0;
        this.cache = new HashMap<>();
        this.frequencyMap = new HashMap<>();
    }

    public V get(K key) {
        if (!cache.containsKey(key)) {
            return null;
        }

        CacheEntry<V> entry = cache.get(key);
        updateFrequency(key, entry);
        return entry.value;
    }

    public void put(K key, V value) {
        if (capacity <= 0) return;

        if (cache.containsKey(key)) {
            CacheEntry<V> entry = cache.get(key);
            entry.value = value;
            updateFrequency(key, entry);
            return;
        }

        // Evict if at capacity
        if (cache.size() >= capacity) {
            evictLeastFrequent();
        }

        // Add new entry with frequency 1
        CacheEntry<V> newEntry = new CacheEntry<>(value, 1);
        cache.put(key, newEntry);
        frequencyMap.computeIfAbsent(1, k -> new LinkedHashSet<>
()).add(key);
        minFrequency = 1;
    }

    private void updateFrequency(K key, CacheEntry<V> entry) {
        int oldFreq = entry.frequency;
        int newFreq = oldFreq + 1;

        // Remove from old frequency bucket
        LinkedHashSet<K> oldBucket = frequencyMap.get(oldFreq);
        oldBucket.remove(key);

        // Update min frequency if bucket is now empty
        if (oldBucket.isEmpty() && minFrequency == oldFreq) {
            minFrequency = newFreq;
        }

        // Add to new frequency bucket
        frequencyMap.computeIfAbsent(newFreq, k -> new LinkedHashSet<>
()).add(key);
        entry.frequency = newFreq;
    }

    private void evictLeastFrequent() {
        LinkedHashSet<K> minFreqBucket = frequencyMap.get(minFrequency);
        if (minFreqBucket != null && !minFreqBucket.isEmpty()) {
            // Remove first (oldest) entry at minimum frequency
            K evictKey = minFreqBucket.iterator().next();
            minFreqBucket.remove(evictKey);
            cache.remove(evictKey);
        }
    }

    private static class CacheEntry<V> {
        V value;
        int frequency;

        CacheEntry(V value, int frequency) {
            this.value = value;
            this.frequency = frequency;
        }
    }
}
```

3. FIFO (First In First Out)

- Simplest: Evict oldest entry
- Doesn't consider access patterns

4. TTL (Time To Live)

```
// Expire after 1 hour
cache.set(key, value, Duration.ofHours(1));
```

Advanced Caching Techniques

1. Bloom Filters (Negative Cache)

```
import com.google.common.hash.BloomFilter;
import com.google.common.hash.Funnels;

/**
 * Bloom filter for avoiding unnecessary DB queries.
 * False positives possible, but no false negatives.
 */
public class UserCacheWithBloomFilter {

    // Bloom filter with 1M expected insertions and 1% false positive rate
    private final BloomFilter<String> bloomFilter =
        BloomFilter.create(Funnels.stringFunnel(StandardCharsets.UTF_8),
                           1_000_000, 0.01);

    private final Cache<String, User> cache;
    private final UserRepository db;

    public Optional<User> getUser(String userId) {
        // Check bloom filter first - O(1) operation
        if (!bloomFilter.mightContain(userId)) {
            return Optional.empty();  // Definitely doesn't exist
        }

        // Might exist - check cache then DB
        return cacheAsideGet(userId);
    }

    public User createUser(String userId, UserData data) {
        User user = db.insert(userId, data);
        bloomFilter.put(userId);  // Add to bloom filter
        return user;
    }
}
```

2. Probabilistic Early Expiration (Thundering Herd Prevention)

```
import java.util.Random;
import java.util.concurrent.CompletableFuture;
import java.util.function.Supplier;

/**
```

```
 * Cache with probabilistic early expiration to prevent thundering herd.
 * As expiry approaches, probability of refresh increases.
 */
public class ProbabilisticCache<K, V> {

    private final Cache<K, CacheEntry<V>> cache;
    private final Random random = new Random();

    public V getWithEarlyExpiration(K key, Supplier<V> loader, Duration
ttl) {
        CacheEntry<V> entry = cache.get(key);

        if (entry == null) {
            // Cache miss - load data
            V value = loader.get();
            cache.set(key, new CacheEntry<>(value,
Instant.now().plus(ttl)), ttl);
            return value;
        }

        // Calculate time remaining until expiry
        long remainingMs = Duration.between(Instant.now(),
entry.expiry).toMillis();
        long ttlMs = ttl.toMillis();

        // Probability increases as expiry approaches
        // At half TTL: 50% chance, near expiry: ~100% chance
        double probability = 1.0 - ((double) remainingMs / ttlMs);

        if (random.nextDouble() < probability) {
            // Async refresh to avoid blocking
            CompletableFuture.runAsync(() -> {
                V freshValue = loader.get();
                cache.set(key, new CacheEntry<>(freshValue,
Instant.now().plus(ttl)), ttl);
            });
        }

        return entry.value;
    }

    private record CacheEntry<V>(V value, Instant expiry) {}
}
```

3. Consistent Hashing for Cache Distribution

```
import java.security.MessageDigest;
import java.util.*;

/**
 * Consistent hash ring for distributing cache keys across nodes.
 * Minimizes cache invalidation when nodes are added/removed.
 */
public class ConsistentHashCacheRouter {

    private final TreeMap<String, String> ring = new TreeMap<>();
    private final int virtualNodes;

    public ConsistentHashCacheRouter(List<String> nodes, int virtualNodes)
{
        this.virtualNodes = virtualNodes;
        for (String node : nodes) {
            addNode(node);
        }
    }

    public void addNode(String node) {
        for (int i = 0; i < virtualNodes; i++) {
            String hash = computeMD5Hash(node + ":" + i);
            ring.put(hash, node);
        }
    }

    public void removeNode(String node) {
        for (int i = 0; i < virtualNodes; i++) {
            String hash = computeMD5Hash(node + ":" + i);
            ring.remove(hash);
        }
    }

    public String getNodeForKey(String key) {
        if (ring.isEmpty()) {
            return null;
        }

        String hash = computeMD5Hash(key);
        Map.Entry<String, String> entry = ring.ceilingEntry(hash);

        // Wrap around to first node
        return (entry != null) ? entry.getValue() :
ring.firstEntry().getValue();
    }

    private String computeMD5Hash(String input) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] digest = md.digest(input.getBytes());
            StringBuilder sb = new StringBuilder();
            for (byte b : digest) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (Exception e) {
            throw new RuntimeException("MD5 computation failed", e);
        }
    }
}

// Usage example
List<String> cacheNodes = List.of("cache1:6379", "cache2:6379",
"cache3:6379");
ConsistentHashCacheRouter router = new
ConsistentHashCacheRouter(cacheNodes, 150);

public String cacheGet(String key) {
    String node = router.getNodeForKey(key);
    return redisClients.get(node).get(key);
}
```

Monitoring Cache Performance

Key Metrics:

```
/**
 * Cache performance monitoring and metrics calculation.
 */
public class CacheMetrics {

    private final AtomicLong cacheHits = new AtomicLong(0);
    private final AtomicLong cacheMisses = new AtomicLong(0);
    private final AtomicLong evictions = new AtomicLong(0);
    private final AtomicLong totalSets = new AtomicLong(0);
    private final AtomicLong hitsBeforeExpiry = new AtomicLong(0);

    /**
     * Calculate cache hit rate.
     * Target: > 80% for most applications
     */
    public double getCacheHitRate() {
        long total = cacheHits.get() + cacheMisses.get();
        return total > 0 ? (double) cacheHits.get() / total : 0.0;
    }

    /**
     * Calculate eviction rate.
     * High rate indicates cache is too small
     */
    public double getEvictionRate() {
        long totalOps = cacheHits.get() + cacheMisses.get();
        return totalOps > 0 ? (double) evictions.get() / totalOps : 0.0;
    }

    /**
     * Calculate TTL effectiveness.
     * Low rate indicates TTL is too short
     */
    public double getTtlHitRate() {
        long sets = totalSets.get();
        return sets > 0 ? (double) hitsBeforeExpiry.get() / sets : 0.0;
    }

    /**
     * Calculate memory utilization.
     * Target: 70-80% (leave headroom for spikes)
     */
    public double getMemoryUtilization(long usedMemory, long maxMemory) {
        return maxMemory > 0 ? (double) usedMemory / maxMemory : 0.0;
    }

    // Increment methods for tracking
    public void recordHit() { cacheHits.incrementAndGet(); }
    public void recordMiss() { cacheMisses.incrementAndGet(); }
    public void recordEviction() { evictions.incrementAndGet(); }
    public void recordSet() { totalSets.incrementAndGet(); }
    public void recordHitBeforeExpiry() {
hitsBeforeExpiry.incrementAndGet(); }
}
```

---

19. Adapters (File and FTP)

Adapter Pattern Overview

Purpose: Translate between different interfaces or protocols, allowing systems with incompatible interfaces to work together.

File Adapter

Use Case: Read data from local or network file systems (CSV, JSON, XML, TXT).

Implementation:

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.dataformat.xml.*;

/**
 * Abstract base class for file format adapters.
 * Enables reading/writing different file formats through a common
interface.
 */
public abstract class FileAdapter<T> {

    public abstract List<T> read(Path filePath, Class<T> type) throws
IOException;
```

```
    public abstract void write(Path filePath, List<T> data) throws
IOException;
}

/**
 * CSV file adapter using Jackson CSV.
 */
public class CSVAdapter<T> extends FileAdapter<T> {

    private final CsvMapper csvMapper;

    public CSVAdapter() {
        this.csvMapper = new CsvMapper();
    }

    @Override
    public List<T> read(Path filePath, Class<T> type) throws IOException {
        CsvSchema schema = csvMapper.schemaFor(type).withHeader();

        try (Reader reader = Files.newBufferedReader(filePath)) {
            MappingIterator<T> iterator = csvMapper
                .readerFor(type)
                .with(schema)
                .readValues(reader);
            return iterator.readAll();
        }
    }

    @Override
    public void write(Path filePath, List<T> data) throws IOException {
        if (data == null || data.isEmpty()) {
            return;
        }

        CsvSchema schema =
csvMapper.schemaFor(data.get(0).getClass()).withHeader();

        try (Writer writer = Files.newBufferedWriter(filePath)) {
            csvMapper.writer(schema).writeValues(writer).writeAll(data);
        }
    }
}

/**
 * JSON file adapter using Jackson.
 */
public class JSONAdapter<T> extends FileAdapter<T> {

    private final ObjectMapper objectMapper;

    public JSONAdapter() {
        this.objectMapper = new ObjectMapper()
            .enable(SerializationFeature.INDENT_OUTPUT);
    }
    @Override
    public List<T> read(Path filePath, Class<T> type) throws IOException {
        JavaType listType = objectMapper.getTypeFactory()
            .constructCollectionType(List.class, type);
        return objectMapper.readValue(filePath.toFile(), listType);
    }

    @Override
    public void write(Path filePath, List<T> data) throws IOException {
        objectMapper.writeValue(filePath.toFile(), data);
    }
}

/**
 * XML file adapter using Jackson XML.
 */
public class XMLAdapter<T> extends FileAdapter<T> {

    private final XmlMapper xmlMapper;

    public XMLAdapter() {
        this.xmlMapper = new XmlMapper();
    }

    @Override
    public List<T> read(Path filePath, Class<T> type) throws IOException {
        JavaType listType = xmlMapper.getTypeFactory()
            .constructCollectionType(List.class, type);
        return xmlMapper.readValue(filePath.toFile(), listType);
    }

    @Override
    public void write(Path filePath, List<T> data) throws IOException {
        xmlMapper.writeValue(filePath.toFile(), data);
    }
}

/**
 * Factory for creating appropriate file adapter based on file type.
 */
public class FileAdapterFactory {

    private static final Map<String, FileAdapter<?>> adapters = new
HashMap<>();

    static {
        adapters.put("csv", new CSVAdapter<>());
        adapters.put("json", new JSONAdapter<>());
        adapters.put("xml", new XMLAdapter<>());
    }

    @SuppressWarnings("unchecked")
    public static <T> FileAdapter<T> getAdapter(String fileType) {
        FileAdapter<?> adapter = adapters.get(fileType.toLowerCase());
        if (adapter == null) {
            throw new IllegalArgumentException("Unsupported file type: " +
fileType);
        }
        return (FileAdapter<T>) adapter;
    }

    public static <T> FileAdapter<T> getAdapterForFile(Path filePath) {
        String fileName = filePath.getFileName().toString();
        String extension = fileName.substring(fileName.lastIndexOf('.') +
1);
        return getAdapter(extension);
    }
}

// Usage example
// FileAdapter<User> adapter = FileAdapterFactory.getAdapter("csv");
// List<User> users = adapter.read(Path.of("data.csv"), User.class);
// adapter.write(Path.of("output.csv"), processedUsers);
```

Advanced File Adapter (Streaming for Large Files):

```
import java.util.*;
import java.util.function.Consumer;
import java.io.*;
import java.nio.file.*;

/**
 * Streaming CSV adapter for processing large files without loading
 * everything into memory.
 */
public class StreamingCSVAdapter<T> {

    private final CsvMapper csvMapper;
    private final Class<T> type;

    public StreamingCSVAdapter(Class<T> type) {
        this.csvMapper = new CsvMapper();
        this.type = type;
    }

    /**
     * Read file in chunks for memory-efficient processing.
     *
     * @param filePath Path to the CSV file
     * @param chunkSize Number of records per chunk
     * @return Iterator of chunks
     */
    public Iterator<List<T>> readInChunks(Path filePath, int chunkSize)
            throws IOException {
```

```
        CsvSchema schema = csvMapper.schemaFor(type).withHeader();
        Reader reader = Files.newBufferedReader(filePath);
        MappingIterator<T> iterator = csvMapper
            .readerFor(type)
            .with(schema)
            .readValues(reader);

        return new Iterator<>() {
            @Override
            public boolean hasNext() {
                return iterator.hasNext();
            }

            @Override
            public List<T> next() {
                List<T> chunk = new ArrayList<>(chunkSize);
                while (iterator.hasNext() && chunk.size() < chunkSize) {
                    chunk.add(iterator.next());
                }
                return chunk;
            }
        };
    }

    /**
     * Process file in streaming fashion with a consumer function.
     */
    public void processStream(Path filePath, int chunkSize,
                              Consumer<List<T>> chunkProcessor) throws
IOException {
        Iterator<List<T>> chunks = readInChunks(filePath, chunkSize);
        while (chunks.hasNext()) {
            chunkProcessor.accept(chunks.next());
        }
    }

    /**
     * Write data from a stream/generator to file without buffering all
data.
     */
    public void writeStream(Path filePath, Iterator<List<T>> dataIterator)
            throws IOException {

        if (!dataIterator.hasNext()) {
            return;
        }

        List<T> firstChunk = dataIterator.next();
        CsvSchema schema = csvMapper.schemaFor(type).withHeader();

        try (Writer writer = Files.newBufferedWriter(filePath)) {
            SequenceWriter seqWriter =
csvMapper.writer(schema).writeValues(writer);
```

```
            // Write first chunk
            for (T item : firstChunk) {
                seqWriter.write(item);
            }

            // Write remaining chunks
            while (dataIterator.hasNext()) {
                for (T item : dataIterator.next()) {
                    seqWriter.write(item);
                }
            }
        }
    }
}

// Usage for large files
// StreamingCSVAdapter<User> adapter = new StreamingCSVAdapter<>
(User.class);
// adapter.processStream(
//     Path.of("large_file.csv"),
//     10000,
//     chunk -> processChunk(chunk)
// );
```

FTP Adapter

Use Case: Transfer files to/from FTP servers, common in legacy system integrations.

Implementation:

```
import org.apache.commons.net.ftp.FTP;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPSClient;
import java.io.*;
import java.nio.file.*;

/**
 * FTP adapter for file transfers with optional TLS support.
 * Implements AutoCloseable for try-with-resources usage.
 */
public class FTPAdapter implements AutoCloseable {

    private final String host;
    private final String username;
    private final String password;
    private final int port;
    private final boolean useTls;
    private FTPClient ftpClient;

    public FTPAdapter(String host, String username, String password,
                      int port, boolean useTls) {
```

```
        this.host = host;
        this.username = username;
        this.password = password;
        this.port = port;
        this.useTls = useTls;
    }

    public FTPAdapter connect() throws IOException {
        ftpClient = useTls ? new FTPSClient() : new FTPClient();
        ftpClient.connect(host, port);
        ftpClient.login(username, password);
        ftpClient.enterLocalPassiveMode();
        ftpClient.setFileType(FTP.BINARY_FILE_TYPE);

        if (useTls) {
            ((FTPSClient) ftpClient).execPBSZ(0);
            ((FTPSClient) ftpClient).execPROT("P");
        }
        return this;
    }

    public void disconnect() {
        if (ftpClient != null && ftpClient.isConnected()) {
            try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException e) {
                // Log and ignore
            }
        }
    }

    public void upload(Path localPath, String remotePath) throws
IOException {
        try (InputStream input = Files.newInputStream(localPath)) {
            boolean success = ftpClient.storeFile(remotePath, input);
            if (!success) {
                throw new IOException("FTP upload failed: " +
ftpClient.getReplyString());
            }
        }
    }

    public void download(String remotePath, Path localPath) throws
IOException {
        try (OutputStream output = Files.newOutputStream(localPath)) {
            boolean success = ftpClient.retrieveFile(remotePath, output);
            if (!success) {
                throw new IOException("FTP download failed: " +
ftpClient.getReplyString());
            }
        }
    }
    public String[] listFiles(String remoteDir) throws IOException {
        return ftpClient.listNames(remoteDir);
    }

    public void delete(String remotePath) throws IOException {
        ftpClient.deleteFile(remotePath);
    }

    public void createDirectory(String remoteDir) throws IOException {
        ftpClient.makeDirectory(remoteDir);
    }

    @Override
    public void close() {
        disconnect();
    }
}

// Usage
try (FTPAdapter ftp = new FTPAdapter("ftp.example.com", "user", "pass",
21, true).connect()) {
    // Upload file
    ftp.upload(Path.of("local_data.csv"), "/remote/data.csv");

    // List files
    String[] files = ftp.listFiles("/remote");

    // Download file
    ftp.download("/remote/results.csv", Path.of("local_results.csv"));
}
```

Advanced FTP Adapter (Retry, Logging, Progress):

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.function.Supplier;

/**
 * Advanced FTP adapter with retry logic, logging, and progress tracking.
 */
public class AdvancedFTPAdapter extends FTPAdapter {

    private static final Logger logger =
LoggerFactory.getLogger(AdvancedFTPAdapter.class);

    private final int maxRetries;
    private final long retryDelayMs;

    public AdvancedFTPAdapter(String host, String username, String
password,
                              int port, boolean useTls, int maxRetries,
long retryDelayMs) {
        super(host, username, password, port, useTls);
        this.maxRetries = maxRetries;
        this.retryDelayMs = retryDelayMs;
    }

    /**
     * Execute operation with retry logic.
     */
    private <T> T retryOperation(Supplier<T> operation) throws IOException
{
        Exception lastException = null;

        for (int attempt = 1; attempt <= maxRetries; attempt++) {
            try {
                return operation.get();
            } catch (Exception e) {
                lastException = e;
                logger.warn("Attempt {} failed: {}", attempt,
e.getMessage());

                if (attempt < maxRetries) {
                    try {
                        Thread.sleep(retryDelayMs);
                        disconnect();
                        connect();
                    } catch (InterruptedException ie) {
                        Thread.currentThread().interrupt();
                        throw new IOException("Interrupted during retry",
ie);
                    }
                }
            }
        }
        throw new IOException("Operation failed after " + maxRetries + "
attempts", lastException);
    }

    /**
     * Upload with progress callback.
     */
    public void uploadWithProgress(Path localPath, String remotePath,
                                   ProgressCallback callback) throws
IOException {
        long fileSize = Files.size(localPath);
        long uploaded = 0;

        try (InputStream input = new
BufferedInputStream(Files.newInputStream(localPath))) {
            // Custom progress tracking implementation
            retryOperation(() -> {
                super.upload(localPath, remotePath);
                return true;
            });
        }
        if (callback != null) {
            callback.onComplete(fileSize);
        }
    }

    /**
     * Sync local directory to remote.
     */
    public void syncDirectory(Path localDir, String remoteDir) throws
IOException {
        Files.walk(localDir).forEach(localPath -> {
            try {
                Path relativePath = localDir.relativize(localPath);
                String remotePath = remoteDir + "/" +
relativePath.toString().replace("\\", "/");

                if (Files.isDirectory(localPath)) {
                    try {
                        createDirectory(remotePath);
                    } catch (IOException e) {
                        // Directory might already exist
                    }
                } else {
                    logger.info("Uploading {} to {}", localPath,
remotePath);
                    upload(localPath, remotePath);
                }
            } catch (IOException e) {
                logger.error("Failed to sync {}: {}", localPath,
e.getMessage());
            }
        });
    }

    @FunctionalInterface
    public interface ProgressCallback {
        void onComplete(long totalBytes);
    }
}

// Usage
try (AdvancedFTPAdapter ftp = new AdvancedFTPAdapter(
        "ftp.example.com", "user", "pass", 21, true, 3, 5000L)) {
    ftp.connect();
    ftp.syncDirectory(Path.of("/local/data"), "/remote/backup");
}
```

SFTP Adapter (SSH File Transfer)

```
import com.jcraft.jsch.*;
import java.io.*;
import java.util.*;

/**
 * SFTP adapter for secure file transfers using SSH.
 * Supports both password and key-based authentication.
 */
public class SFTPAdapter implements AutoCloseable {

    private final String host;
    private final String username;
    private final String password;
    private final String keyFile;
    private final int port;

    private Session session;
    private ChannelSftp sftpChannel;

    public SFTPAdapter(String host, String username, String password,
                       String keyFile, int port) {
        this.host = host;
        this.username = username;
        this.password = password;
        this.keyFile = keyFile;
        this.port = port;
    }

    public SFTPAdapter connect() throws JSchException {
        JSch jsch = new JSch();

        if (keyFile != null) {
            jsch.addIdentity(keyFile);
        }

        session = jsch.getSession(username, host, port);

        if (password != null) {
            session.setPassword(password);
        }

        session.setConfig("StrictHostKeyChecking", "no");
        session.connect();

        Channel channel = session.openChannel("sftp");
        channel.connect();
        sftpChannel = (ChannelSftp) channel;

        return this;
    }

    public void disconnect() {
        if (sftpChannel != null) sftpChannel.disconnect();
        if (session != null) session.disconnect();
    }

    public void upload(String localPath, String remotePath) throws
SftpException {
        sftpChannel.put(localPath, remotePath);
    }

    public void download(String remotePath, String localPath) throws
SftpException {
        sftpChannel.get(remotePath, localPath);
    }

    public List<String> listFiles(String remoteDir) throws SftpException {
        List<String> files = new ArrayList<>();
        Vector<ChannelSftp.LsEntry> entries = sftpChannel.ls(remoteDir);
        for (ChannelSftp.LsEntry entry : entries) {
```

Use Cases in System Design

1. ETL Pipelines:

```
/**
 * ETL Pipeline: Extract from FTP, Transform, Load to Database.
 */
public class ETLPipeline {

    private final FTPAdapter ftpAdapter;
    private final CSVAdapter csvAdapter;
    private final JdbcTemplate db;

    public void runPipeline() throws Exception {
        // Extract: Download from FTP
        try (FTPAdapter ftp = ftpAdapter.connect()) {
            ftp.download("/data/export.csv", Path.of("temp/export.csv"));
        }

        // Transform: Read and process CSV data
        List<DataRow> data = csvAdapter.read(Path.of("temp/export.csv"),
DataRow.class);
        List<TransformedRow> transformed = transformData(data);

        // Load: Bulk insert into database
        db.batchUpdate("INSERT INTO target_table VALUES (?, ?, ?)",
transformed);
    }
}
```

2. Legacy System Integration:

```
/**
 * Adapter for legacy systems that only support FTP for data exchange.
 */
public class LegacySystemAdapter {

    private final FTPAdapter ftp;
    private final CSVAdapter csvAdapter;

    public LegacySystemAdapter() {
        this.ftp = new FTPAdapter("legacy.ftp.com", "user", "pass", 21,
false);
        this.csvAdapter = new CSVAdapter();
    }

    public void exportOrders(List<Order> orders) throws Exception {
        // Convert to CSV format
        csvAdapter.write(Path.of("orders.csv"), orders);

        // Upload to legacy FTP
        try (FTPAdapter connected = ftp.connect()) {
            connected.upload(Path.of("orders.csv"), "/import/orders.csv");
        }
    }

    public List<Result> importResults() throws Exception {
        // Download from FTP
        try (FTPAdapter connected = ftp.connect()) {
            connected.download("/export/results.csv",
Path.of("results.csv"));
        }

        // Parse and return
        return csvAdapter.read(Path.of("results.csv"), Result.class);
    }
}
```

---

20. Strong vs Eventual Consistency

Strong Consistency

Definition: All clients see the same data at the same time, immediately after a write.

Guarantees:

- Read always returns most recent write
- No stale reads
- Linearizability: Operations appear atomic

Implementation: ACID transactions, distributed consensus (Paxos, Raft)

Example:

```java
/**
 * Bank account transfer - requires strong consistency.
 * Uses ACID transactions to ensure atomicity and isolation.
 */
@Transactional(isolation = Isolation.SERIALIZABLE)
public void transfer(Long fromAccount, Long toAccount, BigDecimal amount)
{
    // Read current balances within transaction
    BigDecimal fromBalance = jdbcTemplate.queryForObject(
        "SELECT balance FROM accounts WHERE id = ? FOR UPDATE",
        BigDecimal.class, fromAccount);
    BigDecimal toBalance = jdbcTemplate.queryForObject(
        "SELECT balance FROM accounts WHERE id = ? FOR UPDATE",
        BigDecimal.class, toAccount);

    // Update balances atomically
    jdbcTemplate.update("UPDATE accounts SET balance = ? WHERE id = ?",
        fromBalance.subtract(amount), fromAccount);
    jdbcTemplate.update("UPDATE accounts SET balance = ? WHERE id = ?",
        toBalance.add(amount), toAccount);

    // Both updates commit atomically
    // No intermediate state visible to other transactions
}
```

Pros:

- Simple programming model
- No data anomalies
- Predictable behavior

Cons:

- Higher latency (coordination required)
- Lower availability (can't tolerate partitions)
- Reduced throughput

Eventual Consistency

Definition: Given enough time without new updates, all replicas will converge to the same state.

Guarantees:

- Reads may return stale data
- Eventually all replicas agree
- High availability during partitions

Implementation: Asynchronous replication, gossip protocols

Example:

```java
/**
 * Social media likes - eventual consistency is acceptable.
 * Fast local write with async replication.
 */
@Service
public class LikeService {

    private final JdbcTemplate localDb;
    private final KafkaTemplate<String, ReplicationEvent>
replicationQueue;

    /**
     * Like a post with eventual consistency.
     */
    public String likePost(Long postId, Long userId) {
        // Write to local datacenter (fast, low latency)
        localDb.update("INSERT INTO likes (post_id, user_id) VALUES (?,
?)",
            postId, userId);

        // Asynchronously replicate to other datacenters
        ReplicationEvent event = ReplicationEvent.builder()
            .operation("insert")
            .table("likes")
            .data(Map.of("post_id", postId, "user_id", userId))
            .build();

        replicationQueue.send("replication-topic", event);

        // Immediate response to user
        return "Liked!";

        // Note: Users in other datacenters might not see this like
immediately
        // But will see it after replication completes (seconds to
minutes)
    }
}
```

Pros:

- Low latency (no coordination)
- High availability (tolerates partitions)
- High throughput

Cons:

- Complex programming model
- Potential data conflicts
- Stale reads

Consistency Models Spectrum

```
Strong ←──────────────────────────────────────────→ Weak

Linearizable          Causal            Eventually
(ACID)               Consistency        Consistent
   │                     │                   │
   │                     │                   │
   ▼                     ▼                   ▼
PostgreSQL          Cassandra          DynamoDB
(default)          (if configured)      (default)

Latency: High                             Low
Availability: Lower                       Higher
```

CAP Theorem

```
        Consistency
             ▲
            /|\
           / | \
          /  |  \
         /   |   \
        /    |    \
       /     |     \
      /  CP  |  CA  \
     /       |       \
    /────────┼────────\
   /    AP   |         \
  /          |          \
Partition ──────────────► Availability
Tolerance

CP: Consistent + Partition Tolerant
   - MongoDB, HBase, Redis
   - Sacrifice availability during partition

CA: Consistent + Available
   - PostgreSQL, MySQL (single node)
   - Not partition tolerant (breaks during network split)

AP: Available + Partition Tolerant
   - Cassandra, DynamoDB, Riak
   - Sacrifice consistency for availability
```

When to Use Each

Use Strong Consistency When:

- Financial transactions (payments, transfers)
- Inventory management (prevent overselling)
- Seat bookings (prevent double booking)
- User authentication
- Regulatory compliance required

Use Eventual Consistency When:

- Social media feeds
- Analytics and metrics
- Product catalogs
- User profiles
- DNS records
- Caching layers

Handling Eventual Consistency

1. Conflict Resolution (Last-Write-Wins):

```java
/**
 * Eventually consistent database with LWW conflict resolution.
 */
public class EventuallyConsistentDB {

    public void write(String key, Object value) {
        long timestamp = System.currentTimeMillis();
        store(key, value, timestamp);
        replicateAsync(key, value, timestamp);
    }

    public <T extends Timestamped> T mergeConflict(T localValue, T
remoteValue) {
        // Resolve by timestamp (Last-Write-Wins)
        if (remoteValue.getTimestamp() > localValue.getTimestamp()) {
            return remoteValue;
        }
        return localValue;
    }
}
```

2. Vector Clocks (Detect Conflicts):

```java
/**
 * Vector clock for tracking causality across replicas.
 */
public class VectorClock {
    // Track writes per replica
    private final Map<String, Long> clock = new ConcurrentHashMap<>();
    // Example: {"replica_1": 5, "replica_2": 3, "replica_3": 7}
    public void increment(String replicaId) {
        clock.merge(replicaId, 1L, Long::sum);
    }

    // Concurrent writes = conflict, application must resolve
    public boolean isConcurrent(VectorClock other) {
        return !this.happensBefore(other) && !other.happensBefore(this);
    }
}
```

3. CRDTs (Conflict-Free Replicated Data Types):

```
/**
 * G-Counter (Grow-only counter) - a CRDT that automatically
 * resolves conflicts without coordination.
 */
public class GCounter {

    private final String replicaId;
    private final Map<String, Long> counts = new ConcurrentHashMap<>();

    public GCounter(String replicaId) {
        this.replicaId = replicaId;
    }

    public void increment() {
        counts.merge(replicaId, 1L, Long::sum);
    }

    public long value() {
        return counts.values().stream().mapToLong(Long::longValue).sum();
    }

    public void merge(GCounter other) {
        other.counts.forEach((replica, count) ->
            counts.merge(replica, count, Math::max));
    }
}

// Automatically resolves conflicts without coordination
```

Trade-offs Summary

Aspect	Strong	Eventual
Consistency	Immediate	Delayed
Latency	Higher	Lower

Aspect	Strong	Eventual
Availability	Lower	Higher
Partition Tolerance	Poor	Good
Complexity	Lower	Higher
Use Case	Critical data	Best-effort data

---

21. Distributed System Consistency

Cross-Region Consistency Challenges

Problem: Maintaining data consistency across geographically distributed datacenters with network latency and potential partitions.

Consistency Patterns

1. Single Master (Asynchronous Replication):

```
          Master (US)
               │
         ┌─────┴─────┐
         │           │
         ▼           ▼
   Replica (EU)  Replica (APAC)

   Writes → Master (low latency for US users)
   Reads → Local replica (low latency globally)
   Replication lag: 100ms - 5s
```

Pros: Simple, fast writes for primary region **Cons**: Stale reads in other regions, single point of failure

2. Multi-Master (Active-Active):

```
Master (US) ←─── Bidirectional ───→ Master (EU)
     │             Replication            │
     └─────────────────┬─────────────────┘
                       │
                Master (APAC)

Writes → Any master (low latency locally)
Conflict resolution required
```

Pros: Low latency globally, high availability **Cons**: Complex conflict resolution

3. Quorum-Based (Consensus):

```
Write requires W replicas to acknowledge
Read requires R replicas to respond

Strong consistency when: R + W > N
(N = total replicas)

Example: N=3, W=2, R=2
   │
   ├─ Replica 1 (US)
   ├─ Replica 2 (EU)     } Any 2 must acknowledge
   └─ Replica 3 (APAC)

Latency: Median of RTT to 2 closest replicas
```

Pros: Tunable consistency/availability **Cons**: Increased latency for coordination

Implementation Strategies

1. Two-Phase Commit (2PC):

```
/**
 * Two-Phase Commit coordinator for distributed transactions.
 */
public class TwoPhaseCommit {

    private final List<TransactionParticipant> participants;

    public TwoPhaseCommit(List<TransactionParticipant> participants) {

        # Check if all prepared
        if all(result == 'PREPARED' for result in prepare_results):
            # Phase 2: Commit
            for participant in self.participants:
                participant.commit(transaction)
            return 'COMMITTED'
        else:
            # Abort
            for participant in self.participants:
                participant.abort(transaction)
            return 'ABORTED'
```

Problem: Blocking protocol, single point of failure (coordinator)

2. Three-Phase Commit (3PC):

Adds pre-commit phase to reduce blocking, but still susceptible to partitions.

3. Paxos / Raft (Consensus Algorithms):

```
Leader Election:
1. Nodes vote for leader
2. Majority required
3. Leader coordinates all writes

Replication:
1. Leader receives write
2. Replicates to followers
3. Waits for majority acknowledgment
4. Commits locally
5. Notifies followers to commit
```

4. Saga Pattern (Long-Running Transactions):

```java
import java.util.*;

/**
 * Saga orchestrator for long-running distributed transactions.
 * Implements compensating transactions for rollback.
 */
public class Saga {

    private final List<SagaStep> steps = new ArrayList<>();

    public void addStep(Runnable action, Runnable compensation) {
        steps.add(new SagaStep(action, compensation));
    }

    public void execute() throws SagaExecutionException {
        List<SagaStep> executedSteps = new ArrayList<>();

        try {
            for (SagaStep step : steps) {
                step.action.run();
                executedSteps.add(step);
            }
        } catch (Exception e) {
            // Rollback: Execute compensations in reverse order
            Collections.reverse(executedSteps);
            for (SagaStep step : executedSteps) {
                try {
                    step.compensation.run();
                } catch (Exception compEx) {
                    // Log and continue with other compensations
                }
            }
            throw new SagaExecutionException("Saga failed, compensations
executed", e);
        }
    }
    private record SagaStep(Runnable action, Runnable compensation) {}
}

// Example: E-commerce order saga
public class OrderSaga {

    public void placeOrder(Order order) throws SagaExecutionException {
        Saga saga = new Saga();

        saga.addStep(
            () -> inventoryService.reserve(order.getProductId(),
order.getQuantity()),
            () -> inventoryService.release(order.getProductId(),
order.getQuantity())
        );

        saga.addStep(
            () -> paymentService.charge(order.getUserId(),
order.getAmount()),
            () -> paymentService.refund(order.getUserId(),
order.getAmount())
        );

        saga.addStep(
            () -> shippingService.createShipment(order.getId()),
            () -> shippingService.cancelShipment(order.getId())
        );

        saga.execute();
    }
}
```

Conflict Resolution Strategies

1. Last-Write-Wins (LWW):

```
/**
 * Last-Write-Wins conflict resolution.
 * Simple but can lose concurrent writes.
 */
public <T extends Timestamped> T resolveConflict(T localDoc, T remoteDoc)
{
    if (remoteDoc.getTimestamp().isAfter(localDoc.getTimestamp())) {
        return remoteDoc;
    }
    return localDoc;
}
```

Issue: Can lose concurrent writes

2. Application-Specific Logic:

```
/**
 * Merge shopping carts using application-specific logic.
 * Takes union of items with max quantity for conflicts.
 */
public Collection<CartItem> resolveShoppingCart(Cart localCart, Cart
remoteCart) {
    Map<Long, CartItem> mergedItems = new HashMap<>();

    // Combine all items from both carts
    Stream.concat(localCart.getItems().stream(),
remoteCart.getItems().stream())
        .forEach(item -> {
            mergedItems.merge(item.getId(), item, (existing, newItem) -> {
                // Keep max quantity on conflict
                existing.setQuantity(Math.max(existing.getQuantity(),
newItem.getQuantity()));
                return existing;
            });
        });

    return mergedItems.values();
}
```

3. CRDTs (Conflict-Free Replicated Data Types):

```
Automatically merge concurrent updates
Examples:
  - G-Counter (increment-only)
  - PN-Counter (increment/decrement)
  - LWW-Register (last-write-wins)
  - OR-Set (observed-remove set)
```

Monitoring Consistency

Metrics to Track:

```
/**
 * Consistency monitoring service for distributed databases.
 */
@Component
public class ConsistencyMonitor {

    private final MeterRegistry metricsRegistry;

    /**
     * Calculate replication lag between master and replica.
     * Alert if > 5 seconds.
     */
    public Duration getReplicationLag(Instant masterTimestamp, Instant
replicaTimestamp) {
        Duration lag = Duration.between(replicaTimestamp,
masterTimestamp);
        metricsRegistry.gauge("replication.lag.seconds",
lag.getSeconds());
        return lag;
    }

    /**
     * Check consistency between master and replica counts.
     * Alert if difference exceeds threshold.
     */
    public ConsistencyCheckResult checkConsistency(String tableName, long
threshold) {
        long masterCount = masterDb.count(tableName);
        long replicaCount = replicaDb.count(tableName);
        long difference = Math.abs(masterCount - replicaCount);

        metricsRegistry.gauge("consistency.difference." + tableName,
difference);

        return new ConsistencyCheckResult(difference <= threshold,
difference);
    }

    /**
     * Track conflict rate for monitoring dashboard.
     */
    public double calculateConflictRate(long conflictsDetected, long
totalWrites) {
        double rate = (double) conflictsDetected / totalWrites;
        metricsRegistry.gauge("conflict.rate", rate);
        return rate;
    }
}
```

---

22. Rate Limiter

Overview

Limit the number of requests a client can make to prevent abuse, ensure fair resource allocation, and protect backend services from overload.

Rate Limiting Algorithms

1. Token Bucket

Concept: Bucket holds tokens. Each request consumes a token. Tokens refill at constant rate.

```java
import java.util.concurrent.atomic.AtomicLong;

/**
 * Token Bucket rate limiter implementation.
 * Allows bursts up to bucket capacity, then rate-limits to refill rate.
 */
public class TokenBucket {

    private final int capacity;
    private final double refillRate; // tokens per second
    private double tokens;
    private long lastRefillTime;

    public TokenBucket(int capacity, double refillRate) {
        this.capacity = capacity;
        this.refillRate = refillRate;
        this.tokens = capacity;
        this.lastRefillTime = System.nanoTime();
    }

    public synchronized boolean allowRequest() {
        refill();
        if (tokens >= 1) {
            tokens -= 1;
            return true;
        }
        return false;
    }

    private void refill() {
        long now = System.nanoTime();
        double elapsedSeconds = (now - lastRefillTime) / 1_000_000_000.0;
        double tokensToAdd = elapsedSeconds * refillRate;
        tokens = Math.min(capacity, tokens + tokensToAdd);
        lastRefillTime = now;
    }
}

// Usage
TokenBucket limiter = new TokenBucket(100, 10); // 100 tokens, 10/sec
refill

if (limiter.allowRequest()) {
    processRequest();
} else {
    return ResponseEntity.status(429).body("Rate limit exceeded");
}
```

Pros: Smooth rate limiting, allows bursts up to capacity **Cons**: Memory per bucket (per user/IP)

2. Leaky Bucket

Concept: Requests enter a queue (bucket). Processed at constant rate. Overflow drops requests.

```java
import java.util.concurrent.ConcurrentLinkedDeque;

/**
 * Leaky Bucket rate limiter implementation.
 * Processes requests at a constant rate, queues up to capacity.
 */
public class LeakyBucket {

    private final int capacity;
    private final double leakRate; // requests per second
    private final ConcurrentLinkedDeque<Long> queue;
    private long lastLeakTime;

    public LeakyBucket(int capacity, double leakRate) {
        this.capacity = capacity;
        this.leakRate = leakRate;
        this.queue = new ConcurrentLinkedDeque<>();
        this.lastLeakTime = System.nanoTime();
    }

    public synchronized boolean allowRequest() {
        leak();
        if (queue.size() < capacity) {
            queue.addLast(System.nanoTime());
            return true;
        }
        return false;
    }

    private void leak() {
        long now = System.nanoTime();
        double elapsedSeconds = (now - lastLeakTime) / 1_000_000_000.0;
        int leaks = (int) (elapsedSeconds * leakRate);

        for (int i = 0; i < Math.min(leaks, queue.size()); i++) {
            queue.pollFirst();
        }

        lastLeakTime = now;
    }
}
```

Pros: Smooth output rate, prevents spikes **Cons**: Can queue requests (latency)

3. Fixed Window Counter

```java
/**
 * Fixed Window Counter rate limiter.
 * Simple but can allow bursts at window boundaries.
 */
public class FixedWindowCounter {

    private final int limit;
    private final long windowSeconds;
    private int count;
    private long windowStart;

    public FixedWindowCounter(int limit, long windowSeconds) {
        this.limit = limit;
        this.windowSeconds = windowSeconds;
        this.count = 0;
        this.windowStart = System.currentTimeMillis() / 1000;
    }

    public synchronized boolean allowRequest() {
        long now = System.currentTimeMillis() / 1000;

        // Reset window if expired
        if (now - windowStart >= windowSeconds) {
            count = 0;
            windowStart = now;
        }

        if (count < limit) {
            count++;
            return true;
        }
        return false;
    }
}
```

Pros: Simple, low memory **Cons**: Burst at window boundaries (100 req at 0:59, 100 at 1:00 = 200/min)

4. Sliding Window Log

```
import java.util.concurrent.ConcurrentLinkedDeque;

/**
 * Sliding Window Log rate limiter.
 * Accurate but memory grows with request count.
 */
public class SlidingWindowLog {

    private final int limit;
    private final long windowSeconds;
    private final ConcurrentLinkedDeque<Long> requests; // Timestamps

    public SlidingWindowLog(int limit, long windowSeconds) {
        this.limit = limit;
        this.windowSeconds = windowSeconds;
        this.requests = new ConcurrentLinkedDeque<>();
    }

    public synchronized boolean allowRequest() {
        long now = System.currentTimeMillis();
        long cutoff = now - (windowSeconds * 1000);

        // Remove expired entries
        while (!requests.isEmpty() && requests.peekFirst() < cutoff) {
            requests.pollFirst();
        }

        if (requests.size() < limit) {
            requests.addLast(now);
            return true;
        }
        return false;
    }
}
```

Pros: Accurate, no boundary issues **Cons**: Memory grows with request count

5. Sliding Window Counter (Redis)

```
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import java.util.*;

/**
 * Redis-based Sliding Window rate limiter.
 * Uses Lua script for atomic operations.
 */
public class SlidingWindowRedis {

    private final RedisTemplate<String, String> redis;
    private final int limit;
    private final long windowSeconds;

    // Lua script for atomic rate limiting
    private static final String LUA_SCRIPT = """
        local key = KEYS[1]
        local now = tonumber(ARGV[1])
        local window_start = tonumber(ARGV[2])
        local limit = tonumber(ARGV[3])
        local window_seconds = tonumber(ARGV[4])

        -- Remove old entries
        redis.call('ZREMRANGEBYSCORE', key, 0, window_start)

        -- Count current requests
        local count = redis.call('ZCARD', key)
        if count < limit then
            redis.call('ZADD', key, now, now)
            redis.call('EXPIRE', key, window_seconds)
            return 1
        else
            return 0
        end
        """;

    public SlidingWindowRedis(RedisTemplate<String, String> redis, int
limit, long windowSeconds) {
        this.redis = redis;
        this.limit = limit;
        this.windowSeconds = windowSeconds;
    }

    public boolean allowRequest(String userId) {
        String key = "rate_limit:" + userId;
        long now = System.currentTimeMillis();
        long windowStart = now - (windowSeconds * 1000);

        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(LUA_SCRIPT, Long.class);

        Long result = redis.execute(
            script,
            Collections.singletonList(key),
            String.valueOf(now),
            String.valueOf(windowStart),
            String.valueOf(limit),
            String.valueOf(windowSeconds)
        );

        return result != null && result == 1;
    }
}
```

Distributed Rate Limiting

Challenge: Multiple API servers need shared rate limit state.

Solution 1: Centralized Counter (Redis):

```
/**
 * Distributed Rate Limiter using Redis for shared state across servers.
 */
@Component
public class DistributedRateLimiter {

    private final RedisTemplate<String, String> redis;

    public DistributedRateLimiter(RedisTemplate<String, String> redis) {
        this.redis = redis;
    }

    public boolean checkRateLimit(String key, int limit, int
windowSeconds) {
        long now = System.currentTimeMillis() / 1000;
        String windowKey = key + ":" + (now / windowSeconds);

        Long count = redis.opsForValue().increment(windowKey);
        if (count == 1) {
            redis.expire(windowKey, Duration.ofSeconds(windowSeconds *
2));
        }

        return count != null && count <= limit;
    }
}

// Usage across multiple servers
if (!limiter.checkRateLimit("user:" + userId, 100, 60)) {
    return ResponseEntity.status(429).body("Rate limit exceeded");
}
```

Solution 2: Distributed Token Bucket:

```
/**
 * Distributed Token Bucket using Redis with Lua script for atomicity.
 */
public class DistributedTokenBucket {

    private final RedisTemplate<String, String> redis;

    private static final String LUA_SCRIPT = """
        local key = KEYS[1]
        local capacity = tonumber(ARGV[1])
        local refill_rate = tonumber(ARGV[2])
        local now = tonumber(ARGV[3])

        local bucket = redis.call('HMGET', key, 'tokens', 'last_refill')
        local tokens = tonumber(bucket[1]) or capacity
        local last_refill = tonumber(bucket[2]) or now

        -- Refill tokens
        local elapsed = now - last_refill
        local new_tokens = math.min(capacity, tokens + (elapsed *
refill_rate))

        if new_tokens >= 1 then
            redis.call('HMSET', key, 'tokens', new_tokens - 1,
'last_refill', now)
            redis.call('EXPIRE', key, 3600)
            return 1
        else
            redis.call('HMSET', key, 'tokens', new_tokens, 'last_refill',
now)
            return 0
        end
        """;

    public DistributedTokenBucket(RedisTemplate<String, String> redis) {
        this.redis = redis;
    }

    public boolean allowRequest(String userId, int capacity, double
refillRate) {
        String key = "token_bucket:" + userId;
        DefaultRedisScript<Long> script = new DefaultRedisScript<>
(LUA_SCRIPT, Long.class);

        Long result = redis.execute(script,
            Collections.singletonList(key),
            String.valueOf(capacity),
            String.valueOf(refillRate),
            String.valueOf(System.currentTimeMillis() / 1000.0));

        return result != null && result == 1;
    }
}
```

Tiered Rate Limiting

```
/**
 * Tiered rate limiter with different limits based on subscription tier.
 */
public class TieredRateLimiter {

    private final DistributedRateLimiter rateLimiter;
    private final Map<String, RateLimitConfig> tierLimits = Map.of(
        "free", new RateLimitConfig(100, 3600),      // 100/hour
        "basic", new RateLimitConfig(1000, 3600),    // 1000/hour
        "premium", new RateLimitConfig(10000, 3600)  // 10000/hour
    );

    public TieredRateLimiter(DistributedRateLimiter rateLimiter) {
        this.rateLimiter = rateLimiter;
    }

    public boolean checkLimit(String userId, String tier) {
        RateLimitConfig config = tierLimits.getOrDefault(tier,
tierLimits.get("free"));
        String key = "user:" + userId + ":" + tier;
        return rateLimiter.checkRateLimit(key, config.requests(),
config.windowSeconds());
    }

    private record RateLimitConfig(int requests, int windowSeconds) {}
}
```

Rate Limiting by Multiple Dimensions

```
/**
 * Multi-dimension rate limiter checking multiple limits per request.
 */
public class MultiDimensionRateLimiter {

    private final DistributedRateLimiter rateLimiter;

    public MultiDimensionRateLimiter(DistributedRateLimiter rateLimiter) {
        this.rateLimiter = rateLimiter;
    }

    public RateLimitResult allowRequest(String userId, String apiKey,
String ipAddress) {
        // Check multiple limits
        List<RateLimitCheck> checks = List.of(
            new RateLimitCheck("user", userId, 1000, 60),     // 1000/min
per user
            new RateLimitCheck("api_key", apiKey, 5000, 60),  // 5000/min
per API key
            new RateLimitCheck("ip", ipAddress, 100, 60),     // 100/min
per IP
            new RateLimitCheck("global", "all", 50000, 60)    // 50000/min
globally
        );

        for (RateLimitCheck check : checks) {
            String key = check.dimension() + ":" + check.key();
            if (!rateLimiter.checkRateLimit(key, check.limit(),
check.windowSeconds())) {
                return new RateLimitResult(false, "Rate limit exceeded for
" + check.dimension());
            }
        }

        return new RateLimitResult(true, null);
    }

    private record RateLimitCheck(String dimension, String key, int limit,
int windowSeconds) {}
    public record RateLimitResult(boolean allowed, String reason) {}
}
```

Response Headers

```
/**
 * Utility to add standard rate limit headers to HTTP responses.
 */
public class RateLimitHeaderUtils {

    public static void addRateLimitHeaders(HttpServletResponse response,
                                            int remaining, int limit, long
resetTime) {
        response.setHeader("X-RateLimit-Limit", String.valueOf(limit));
        response.setHeader("X-RateLimit-Remaining",
String.valueOf(remaining));
        response.setHeader("X-RateLimit-Reset",
String.valueOf(resetTime));

        if (remaining == 0) {
            long retryAfter = resetTime - (System.currentTimeMillis() /
1000);
            response.setHeader("Retry-After", String.valueOf(Math.max(0,
retryAfter)));
        }
    }
}
```

Trade-offs Summary

Algorithm	Pros	Cons	Use Case
Token Bucket	Allows bursts	Memory per user	API gateways
Leaky Bucket	Smooth output	Queue latency	Traffic shaping
Fixed Window	Simple	Burst at edges	Basic limits
Sliding Window	Accurate	More memory	Fair limiting
Distributed	Consistent	Redis dependency	Multi-server

---

23. Top K Heavy Hitter

Problem Overview

Identify the top K most frequent items (heavy hitters) in a massive stream of data with low latency and memory constraints.

Use Cases:

- Top K trending hashtags
- Most visited URLs
- Top IP addresses (DDoS detection)

- Most played songs
- Frequent search queries

Algorithms

1. Exact Count (Hash Map)

```
from collections import Counter
import heapq

class ExactTopK:
    def __init__(self, k):
        self.k = k
        self.counts = Counter()

    def add(self, item):
        self.counts[item] += 1

    def get_top_k(self):
        return heapq.nlargest(self.k, self.counts.items(), key=lambda x:
x[1])

# Example
topk = ExactTopK(k=10)
for item in stream:
    topk.add(item)

top_10 = topk.get_top_k()
```

Memory: O(n) where n = number of unique items **Accuracy**: 100% **Problem**: Not scalable for billions of unique items

2. Count-Min Sketch (Probabilistic)

```
import java.util.*;

/**
 * Count-Min Sketch for approximate frequency counting.
 * Space-efficient probabilistic data structure.
 */
public class CountMinSketch {

    private final int width;
    private final int depth;
    private final long[][] table;

    public CountMinSketch(int width, int depth) {
        this.width = width;
        this.depth = depth;
        this.table = new long[depth][width];
    }

    private int hash(String item, int seed) {
        return Math.abs((item.hashCode() ^ seed) % width);
    }

    public void add(String item, int count) {
        for (int i = 0; i < depth; i++) {
            table[i][hash(item, i)] += count;
        }
    }

    public void add(String item) { add(item, 1); }

    public long estimate(String item) {
        long min = Long.MAX_VALUE;
        for (int i = 0; i < depth; i++) {
            min = Math.min(min, table[i][hash(item, i)]);
        }
        return min;
    }
}

/**
 * Top-K Heavy Hitter using Count-Min Sketch + Min-Heap.
 */
public class TopKHeavyHitter {

    private final int k;
    private final CountMinSketch cms;
    private final PriorityQueue<ItemCount> minHeap;
    private final Set<String> itemsInHeap;

    public TopKHeavyHitter(int k, int width, int depth) {
        this.k = k;
        this.cms = new CountMinSketch(width, depth);
        this.minHeap = new PriorityQueue<>(Comparator.comparingLong(ic ->
ic.count));
        this.itemsInHeap = new HashSet<>();
    }

    public void add(String item) {
        cms.add(item);
        long count = cms.estimate(item);

        if (itemsInHeap.contains(item)) {
            minHeap.removeIf(ic -> ic.item.equals(item));
            minHeap.offer(new ItemCount(item, count));
        } else if (minHeap.size() < k) {
            minHeap.offer(new ItemCount(item, count));
            itemsInHeap.add(item);
        } else if (count > minHeap.peek().count) {
            ItemCount evicted = minHeap.poll();
            itemsInHeap.remove(evicted.item);
```

```
                minHeap.offer(new ItemCount(item, count));
                itemsInHeap.add(item);
            }
        }

    public List<ItemCount> getTopK() {
        return minHeap.stream()
                .sorted(Comparator.comparingLong((ItemCount ic) ->
ic.count).reversed())
                .toList();
    }

    public record ItemCount(String item, long count) {}
}

// Usage
TopKHeavyHitter hh = new TopKHeavyHitter(100, 100000, 7);
for (String item : stream) {
    hh.add(item);
}
List<TopKHeavyHitter.ItemCount> top100 = hh.getTopK();
```

Memory: O(width × depth + k) = O(1) for fixed parameters **Accuracy**: Approximate, with error ε = e / width **Advantage**: Fixed memory regardless of stream size

3. Lossy Counting

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

/**
 * Lossy Counting algorithm for finding frequent items.
 * Guarantees no false negatives for items above threshold.
 */
public class LossyCounting {

    private final double support;
    private final double error;
    private final int bucketWidth;
    private int currentBucket = 1;
    private final Map<String, CountDelta> counts = new HashMap<>();
    private long n = 0;

    public LossyCounting(double supportThreshold, double error) {
        this.support = supportThreshold;
        this.error = error;
        this.bucketWidth = (int) (1.0 / error);
    }

    public void add(String item) {
        n++;
        if (counts.containsKey(item)) {
            CountDelta cd = counts.get(item);
            counts.put(item, new CountDelta(cd.count + 1, cd.delta));
        } else {
            counts.put(item, new CountDelta(1, currentBucket - 1));
        }

        if (n % bucketWidth == 0) {
            currentBucket++;
            prune();
        }
    }

    private void prune() {
        counts.entrySet().removeIf(e ->
            e.getValue().count + e.getValue().delta <= currentBucket);
    }

    public List<Map.Entry<String, Long>> getFrequentItems() {
        double threshold = support * n;
        return counts.entrySet().stream()
            .filter(e -> e.getValue().count >= threshold)
            .map(e -> Map.entry(e.getKey(), (long) e.getValue().count))
            .collect(Collectors.toList());
    }

    private record CountDelta(int count, int delta) {}
}
```

Memory: O(1/ε) where ε = error threshold **Accuracy**: Guarantees: no false negatives, but possible false positives

4. Space-Saving Algorithm

```
import java.util.*;

/**
 * Space-Saving algorithm for finding frequent items.
 * Uses exactly K counters to track potential heavy hitters.
 */
public class SpaceSaving {

    private final int k;
    private final Map<String, Long> counters = new HashMap<>();
    private final PriorityQueue<ItemCount> minHeap;

    public SpaceSaving(int k) {
        this.k = k;
        this.minHeap = new PriorityQueue<>(Comparator.comparingLong(ic ->
ic.count));
```

```
    }

    public void add(String item) {
        if (counters.containsKey(item)) {
            // Increment existing counter
            counters.merge(item, 1L, Long::sum);
        } else if (counters.size() < k) {
            // Add new counter
            counters.put(item, 1L);
            minHeap.offer(new ItemCount(item, 1L));
        } else {
            // Replace minimum counter
            ItemCount minItem = minHeap.poll();
            counters.remove(minItem.item);
            long newCount = minItem.count + 1;
            counters.put(item, newCount);
            minHeap.offer(new ItemCount(item, newCount));
        }
    }

    public List<Map.Entry<String, Long>> getTopK() {
        return counters.entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
            .toList();
    }

    private record ItemCount(String item, long count) {}
}
```

Memory: O(k) **Accuracy**: Guarantees top k items within error bound

Distributed Top K

MapReduce Approach:

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

/**
 * Distributed Top-K using MapReduce pattern.
 */
public class DistributedTopK {

    /**
     * Map phase: Each worker maintains local top K.
     */
    public static class Mapper {
        private final TopKHeavyHitter localTopK;

        public Mapper(int k) {
            this.localTopK = new TopKHeavyHitter(k, 10000, 7);
        }

        public List<TopKHeavyHitter.ItemCount> processChunk(List<String>
dataChunk) {
            for (String item : dataChunk) {
                localTopK.add(item);
            }
            return localTopK.getTopK();
        }
    }

    /**
     * Reduce phase: Merge local top K results.
     */
    public static class Reducer {
        private final int k;
        private final Map<String, Long> globalCounts = new HashMap<>();

        public Reducer(int k) {
            this.k = k;
        }

        public List<Map.Entry<String, Long>> merge(
                List<List<TopKHeavyHitter.ItemCount>> localResults) {
            for (List<TopKHeavyHitter.ItemCount> topKList : localResults)
{
                for (TopKHeavyHitter.ItemCount item : topKList) {
                    globalCounts.merge(item.item(), item.count(),
Long::sum);
                }
            }

            return globalCounts.entrySet().stream()
                .sorted(Map.Entry.<String,
Long>comparingByValue().reversed())
                .limit(k)
                .toList();
        }
    }
}

// Usage
int numWorkers = Runtime.getRuntime().availableProcessors();
List<DistributedTopK.Mapper> mappers = new ArrayList<>();
for (int i = 0; i < numWorkers; i++) {
    mappers.add(new DistributedTopK.Mapper(100));
}
DistributedTopK.Reducer reducer = new DistributedTopK.Reducer(100);

// Process chunks in parallel and merge results
List<List<TopKHeavyHitter.ItemCount>> localResults = // parallel
processing
List<Map.Entry<String, Long>> globalTop100 = reducer.merge(localResults);
```

Real-Time Log Aggregation

Architecture:

```
Logs → Kafka → Stream Processor → Count-Min Sketch → Top K
                 (Flink/Spark)         (State)

Stream Processor:
1. Partition by log type
2. Windowed aggregation (5 min tumbling window)
3. Update Count-Min Sketch
4. Extract Top K every window
5. Publish to Redis/DB

Query Service:
- Read current Top K from Redis
- Latency < 100ms
```

Implementation (Spark Streaming - Java):

```java
import org.apache.spark.streaming.*;
import org.apache.spark.streaming.api.java.*;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import java.util.*;

/**
 * Real-time Top-K aggregation using Spark Streaming.
 */
public class TopKStreamProcessor {

    public static void main(String[] args) {
        // Create streaming context with 1-minute batch interval
        JavaStreamingContext jssc = new JavaStreamingContext(sparkConf,
            Durations.minutes(1));
        jssc.checkpoint("hdfs://checkpoint/topk");

        // Stream from Kafka
        JavaInputDStream<ConsumerRecord<String, String>> kafkaStream =
            KafkaUtils.createDirectStream(jssc,
                LocationStrategies.PreferConsistent(),
                ConsumerStrategies.Subscribe(Collections.singleton("log-
topic"),
                    kafkaParams));

        // Extract URLs from logs
        JavaDStream<String> urls = kafkaStream.map(record ->
            extractUrl(record.value()));
        // Maintain state for top K using updateStateByKey
        JavaPairDStream<String, TopKHeavyHitter> topKState = urls
            .mapToPair(url -> new Tuple2<>("global", url))
            .updateStateByKey((newValues, state) -> {
                TopKHeavyHitter topk = state.orElse(new 
TopKHeavyHitter(100, 10000, 7));
                for (String value : newValues) {
                    topk.add(value);
                }
                return Optional.of(topk);
            });

        // Output to Redis every minute
        topKState.foreachRDD(rdd -> {
            rdd.foreach(tuple -> {
                String key = "topk:" + tuple._1();
                String value = 
objectMapper.writeValueAsString(tuple._2().getTopK());
                redis.set(key, value);
            });
        });

        jssc.start();
        jssc.awaitTermination();
    }

    private static String extractUrl(String logLine) {
        // Parse URL from log line
        return logLine.split(" ")[6]; // Assuming standard log format
    }
}
```

Trade-offs

Algorithm	Memory	Accuracy	Latency	Use Case
Exact Count	O(n)	100%	High	Small datasets
Count-Min	O(1)	~99%	Low	Massive streams
Lossy Counting	O(1/ε)	Guaranteed	Medium	Frequent items
Space-Saving	O(k)	Bounded	Low	Top K only

Recommendations

- **Use Exact Count** for < 1M unique items
- **Use Count-Min Sketch** for billions of items with acceptable 1-2% error
- **Use Space-Saving** when memory is extremely limited
- **Distribute** for throughput > 1M events/sec

---

*Continuing with remaining solutions 24-45...