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
)
```

```sql
-- 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:

```python
# Cache hotels by geohash prefix
def cache_hotels_by_geohash(lat, lon, radius_km):
    geohash = encode(lat, lon, precision=6)  # ~1.2km cell

    # Get adjacent cells for coverage
    neighbors = geohash_neighbors(geohash)

    cache_key = f"hotels:geo:{geohash}"
    cached = redis.get(cache_key)

    if cached:
        return filter_by_distance(cached, lat, lon, radius_km)

    # Cache miss - query DB and cache
    hotels = db.query_by_geohash(geohash)
    redis.setex(cache_key, 3600, hotels)  # 1 hour TTL
    return hotels
```

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
│  u02   │  u08   │  u09   │   + 8 neighbors
└────────┴────────┴────────┘
```

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

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
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:

```python
# Consistent hashing for worker assignment
class ConsistentHash:
    def __init__(self, nodes, virtual_nodes=150):
        self.ring = {}
        self.nodes = nodes
        for node in nodes:
            for i in range(virtual_nodes):
```

```
                key = hashlib.md5(f"{node}:{i}").digest()
                self.ring[key] = node
        self.sorted_keys = sorted(self.ring.keys())

    def get_node(self, url):
        url_hash = hashlib.md5(url).digest()
        for key in self.sorted_keys:
            if url_hash <= key:
                return self.ring[key]
        return self.ring[self.sorted_keys[0]]

# Scheduler main loop
def schedule_urls():
    while True:
        # Fetch due URLs
        urls = db.query("""
            SELECT id, url, frequency
            FROM url_schedule
            WHERE next_run <= NOW()
              AND status = 'pending'
            ORDER BY next_run, priority
            LIMIT 10000
        """)

        # Partition and publish
        for url_record in urls:
            worker = consistent_hash.get_node(url_record.url)
            topic = f"{url_record.frequency}-tasks"
            kafka.publish(topic, url_record, partition_key=worker)

            # Update status
            db.update("""
                UPDATE url_schedule
                SET status = 'processing'
                WHERE id = %s
            """, url_record.id)

        time.sleep(10)  # Poll interval
```

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:

```
// 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:

```python
async def process_payment(request):
    idempotency_key = request.headers.get('Idempotency-Key')

    # Check cache
    cached = await redis.get(f"idempotency:{idempotency_key}")
    if cached:
        return json.loads(cached)  # Return cached result

    # Acquire distributed lock
    lock = await redis.set(
        f"lock:idempotency:{idempotency_key}",
        "1",
        nx=True,
        ex=300  # 5 min expiry
    )

    if not lock:
        # Another request with same key is processing
        await asyncio.sleep(0.1)
        return await redis.get(f"idempotency:{idempotency_key}")

    try:
        # Process payment
        async with db.transaction():
            txn = await db.insert_transaction(request, status='PENDING')
            await kafka.send_transactional(txn)

        # Call provider
        result = await payment_provider.charge(request)

        # Update and cache
        await db.update_transaction(txn.id, result.status)
        await redis.setex(
            f"idempotency:{idempotency_key}",
            86400,  # 24h TTL
            json.dumps(result)
        )

        return result
    finally:
        await redis.delete(f"lock:idempotency:{idempotency_key}")
```

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
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:

```python
# Client sync daemon
class SyncClient:
    def sync_file(self, file_path):
        # 1. Chunk file
        chunks = self.chunk_file(file_path, chunk_size=4*1024*1024)

        # 2. Calculate hashes
        chunk_hashes = [sha256(chunk).hexdigest() for chunk in chunks]

        # 3. Check existing chunks
        response = api.check_chunks(chunk_hashes)
        missing_hashes = response['missing']

        # 4. Upload missing chunks
        for i, chunk_hash in enumerate(chunk_hashes):
```

```
            if chunk_hash in missing_hashes:
                api.upload_chunk(chunk_hash, chunks[i])

        # 5. Create file metadata
        api.create_file(
            name=file_path.name,
            size=sum(len(c) for c in chunks),
            blocks=chunk_hashes
        )

    def watch_changes(self):
        # File system watcher
        watcher = FileSystemWatcher(self.sync_folder)

        # Subscribe to server notifications
        pubsub = redis.subscribe(f"user:{user_id}:changes")

        for event in watcher:
            if event.type == 'created' or event.type == 'modified':
                self.sync_file(event.path)
            elif event.type == 'deleted':
                api.delete_file(event.path)

        # Handle server changes
        for message in pubsub:
            self.download_file(message.file_id)
```

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

Data Type	Storage	Justification
Session State	Redis	Temporary booking holds

Schema:

```sql
-- 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:

```python
class DistributedLock:
    def __init__(self, redis_client):
        self.redis = redis_client

    async def acquire_seat_lock(self, seat_id, session_id, ttl=600):
        # Lua script for atomic check-and-set
        script = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("PEXPIRE", KEYS[1], ARGV[2])
        else
            return redis.call("SET", KEYS[1], ARGV[1], "NX", "PX",
ARGV[2])
        end
        """
        result = await self.redis.eval(
            script,
```

```
            keys=[f"seat:{seat_id}:lock"],
            args=[session_id, ttl * 1000]
        )
        return result == 1 or result == "OK"

    async def release_seat_lock(self, seat_id, session_id):
        # Only release if we own the lock
        script = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("DEL", KEYS[1])
        else
            return 0
        end
        """
        await self.redis.eval(
            script,
            keys=[f"seat:{seat_id}:lock"],
            args=[session_id]
        )

# Booking Service
async def book_seat(user_id, flight_id, seat_id, session_id):
    lock = DistributedLock(redis)

    # 1. Acquire lock
    if not await lock.acquire_seat_lock(seat_id, session_id):
        raise SeatAlreadyHeldError()

    try:
        # 2. Hold seat in DB
        async with db.transaction():
            await db.execute("""
                UPDATE seats
                SET status = 'held',
                    held_until = NOW() + INTERVAL '10 minutes',
                    held_by_session = ?
                WHERE id = ? AND status = 'available'
            """, session_id, seat_id)

            if db.rowcount == 0:
                raise SeatNotAvailableError()

        # 3. Return hold confirmation
        return {"held_until": time.time() + 600}

    except Exception as e:
        # Release lock on error
        await lock.release_seat_lock(seat_id, session_id)
        raise

# Payment completion
async def confirm_booking(booking_id, payment_details):
    booking = await db.get_booking(booking_id)
```

```
    # Idempotency check
    if booking.status == 'confirmed':
        return booking

    # Begin saga
    try:
        # 1. Process payment
        payment_result = await payment_service.charge(payment_details)

        # 2. Confirm booking atomically
        async with db.transaction():
            await db.execute("""
                UPDATE bookings SET status = 'confirmed',
                    payment_id = ?, confirmed_at = NOW()
                WHERE id = ?
            """, payment_result.id, booking_id)

            await db.execute("""
                UPDATE seats SET status = 'booked'
                WHERE id IN (?)
            """, booking.seat_ids)

            await db.execute("""
                UPDATE flights
                SET available_seats = available_seats - ?
                WHERE id = ?
            """, len(booking.seat_ids), booking.flight_id)

        # 3. Sync to aggregators
        await kafka.publish('inventory-updates', {
            'flight_id': booking.flight_id,
            'seats_booked': booking.seat_ids,
            'timestamp': time.time()
        })

        # 4. Release lock
        await lock.release_seat_lock(booking.seat_ids[0],
booking.session_id)

        return booking

    except PaymentError as e:
        # Compensating transaction
        await db.execute("""
            UPDATE seats SET status = 'available', held_until = NULL
            WHERE id IN (?)
        """, booking.seat_ids)
        await lock.release_seat_lock(booking.seat_ids[0],
booking.session_id)
        raise
```

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
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:

```python
# Price update propagation
async def update_price(route_id, provider, price):
    # 1. Update local cache
    await redis.setex(
        f"price:{route_id}:{provider}",
        ttl=300,  # 5 min
        json.dumps(price)
    )

    # 2. Persist to Cassandra (multi-DC)
    await cassandra.execute("""
        INSERT INTO price_snapshots
        (route_id, provider, timestamp, price, currency)
        VALUES (?, ?, ?, ?, ?)
    """, route_id, provider, datetime.now(), price.amount, price.currency)

    # 3. Publish to other DCs via Kafka
    await kafka.publish('price-updates', {
        'route_id': route_id,
        'provider': provider,
        'price': price,
        'dc': 'us-east-1'
    })

# Other DCs consume and update their local cache
async def consume_price_updates():
    async for message in kafka.consume('price-updates'):
        if message.dc != CURRENT_DC:
            await redis.setex(
                f"price:{message.route_id}:{message.provider}",
                ttl=300,
                json.dumps(message.price)
            )
```

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:

```
class GatewayServer:
    def __init__(self):
        self.connections = {}  # user_id -> WebSocket
        self.redis = Redis()

    async def on_connect(self, ws, user_id):
        # Store connection
        self.connections[user_id] = ws

        # Register in Redis (for routing)
        await self.redis.hset('user:gateway', user_id, GATEWAY_ID)
        await self.redis.setex(f'user:{user_id}:online', 60, '1')

        # Deliver pending messages
        pending = await self.fetch_pending_messages(user_id)
        for msg in pending:
            await ws.send(msg)

    async def on_message(self, user_id, message):
        recipient_id = message.recipient_id

        # Find recipient's gateway
        gateway_id = await self.redis.hget('user:gateway', recipient_id)

        if gateway_id:
            # Recipient online - direct delivery
            if gateway_id == GATEWAY_ID:
                # Same gateway
                await self.connections[recipient_id].send(message)
            else:
                # Different gateway - use inter-gateway messaging
                await self.send_to_gateway(gateway_id, message)
        else:
            # Recipient offline - queue message
            await kafka.publish('offline_messages', message)

        # Store in Cassandra (async)
        await cassandra.insert_message(message)

    async def on_disconnect(self, user_id):
        del self.connections[user_id]
        await self.redis.hdel('user:gateway', user_id)
        await self.redis.delete(f'user:{user_id}:online')
```

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:

```
async def cancel_appointment(appointment_id, cancelled_by):
    async with db.transaction():
        # Update appointment status
        await db.execute("""
            UPDATE appointments
            SET status = 'cancelled', cancelled_at = NOW()
            WHERE id = ?
        """, appointment_id)

        # Get appointment details
        appt = await db.get_appointment(appointment_id)

        # Check waitlist
        waitlist = await db.query("""
            SELECT * FROM waitlist
            WHERE doctor_id = ?
              AND preferred_date = ?
            ORDER BY created_at
            LIMIT 1
        """, appt.doctor_id, appt.appointment_date)

        if waitlist:
            # Notify waitlisted patient
            await notification_service.send(
                waitlist.patient_id,
                f"Slot available: {appt.appointment_date}
{appt.start_time}"
            )

            # Auto-book if patient configured
```

```
            if waitlist.auto_book:
                await book_appointment(
                    waitlist.patient_id,
                    appt.doctor_id,
                    appt.appointment_date,
                    appt.start_time
                )
```

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 (
```

```sql
    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:

```python
class DistributedLock:
    def __init__(self, redis_client):
        self.redis = redis_client

    async def acquire(self, lock_key, value, ttl_seconds=30):
        """Acquire lock with automatic expiry"""
        result = await self.redis.set(
            lock_key,
            value,
            nx=True,  # Only set if not exists
            ex=ttl_seconds
        )
        return result is not None

    async def release(self, lock_key, value):
        """Release lock only if we own it"""
        lua_script = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("DEL", KEYS[1])
        else
            return 0
        end
        """
        await self.redis.eval(lua_script, 1, lock_key, value)

    async def extend(self, lock_key, value, ttl_seconds):
        """Extend lock TTL if we own it"""
        lua_script = """
        if redis.call("GET", KEYS[1]) == ARGV[1] then
            return redis.call("EXPIRE", KEYS[1], ARGV[2])
        else
            return 0
        end
        """
        await self.redis.eval(lua_script, 1, lock_key, value, ttl_seconds)

# Booking service
async def book_room(user_id, room_id, check_in, check_out):
    lock_key = f"room:{room_id}:{check_in}:{check_out}"
    session_id = generate_session_id()
    lock = DistributedLock(redis)
```

```
    # Try to acquire lock
    if not await lock.acquire(lock_key, session_id, ttl_seconds=30):
        raise BookingInProgressError("Another user is booking this room")

    try:
        async with db.transaction(isolation='serializable'):
            # Check availability for all nights
            nights = get_date_range(check_in, check_out)
            availability = await db.query("""
                SELECT date, available_rooms
                FROM room_inventory
                WHERE room_id = ? AND date = ANY(?)
                FOR UPDATE
            """, room_id, nights)

            if len(availability) != len(nights):
                raise NoInventoryError("Missing inventory data")

            if any(row['available_rooms'] < 1 for row in availability):
                raise NoAvailabilityError("Room not available")

            # Decrement inventory
            await db.execute("""
                UPDATE room_inventory
                SET available_rooms = available_rooms - 1
                WHERE room_id = ? AND date = ANY(?)
            """, room_id, nights)

            # Create reservation
            reservation_id = await db.insert_reservation(
                user_id, room_id, check_in, check_out
            )

            # Process payment
            payment = await payment_service.charge(user_id, total_price)

            # Update reservation with payment
            await db.execute("""
                UPDATE reservations
                SET status = 'confirmed', payment_id = ?
                WHERE id = ?
            """, payment.id, reservation_id)

            return reservation_id

    except Exception as e:
        # Transaction will auto-rollback
        raise
    finally:
        # Always release lock
        await lock.release(lock_key, session_id)
```

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:

```python
class MultiLevelCache:
    def __init__(self):
        self.local_cache = LRUCache(capacity=1000)
        self.redis = Redis()
        self.db = Database()

    async def get(self, key):
        # L1: Local cache
        value = self.local_cache.get(key)
        if value:
            return value

        # L2: Global cache (Redis)
        value = await self.redis.get(key)
        if value:
            self.local_cache.set(key, value)
            return value

        # L3: Database
        value = await self.db.query(key)
        if value:
            # Populate both caches
            await self.redis.setex(key, 3600, value)  # 1 hour
            self.local_cache.set(key, value)  # In-memory
            return value

        return None
```

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

Aspect	Local Cache	Global Cache	Multi-Level
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:

```python
class ShardRouter:
    def __init__(self, shards):
        self.shards = shards
        self.num_shards = len(shards)

    def get_shard(self, user_id):
        # Hash-based sharding
        shard_id = hash(user_id) % self.num_shards
        return self.shards[shard_id]

    def query(self, user_id):
        shard = self.get_shard(user_id)
        return shard.query(f"SELECT * FROM users WHERE id = {user_id}")

    def query_all_shards(self, query):
        # Fan-out query to all shards
        results = []
        for shard in self.shards:
            results.extend(shard.query(query))
        return results
```

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:

```python
# Each service has its own database
class UserService:
    def __init__(self):
        self.db = connect("user_db")

    def get_user(self, user_id):
        return self.db.query("SELECT * FROM users WHERE id = ?", user_id)
```

```
class OrderService:
    def __init__(self):
        self.db = connect("order_db")

    def get_orders(self, user_id):
        return self.db.query("SELECT * FROM orders WHERE user_id = ?",
user_id)
```

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

```
try:
    orders = order_service.get_orders(user_id)
except ServiceUnavailable:
    orders = []  # Graceful degradation
    log_error("Order service down")
```

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:

```
def get_user(user_id):
    # Try cache first
    user = cache.get(f"user:{user_id}")
    if user:
        return user

    # Cache miss - query DB
    user = db.query("SELECT * FROM users WHERE id = ?", user_id)

    # Populate cache
    cache.set(f"user:{user_id}", user, ttl=3600)
    return user

def update_user(user_id, data):
```

```
    # Update DB
    db.update("UPDATE users SET ... WHERE id = ?", user_id)

    # Invalidate cache
    cache.delete(f"user:{user_id}")
```

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

2. Write-Through Cache

Pattern:

```
def update_user(user_id, data):
    # Write to cache
    cache.set(f"user:{user_id}", data, ttl=3600)

    # Write to DB (synchronously)
    db.update("UPDATE users SET ... WHERE id = ?", user_id)

    return data
```

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

3. Write-Behind (Write-Back) Cache

Pattern:

```
def update_user(user_id, data):
    # Write to cache immediately
    cache.set(f"user:{user_id}", data, ttl=3600)

    # Queue DB write (asynchronously)
    queue.enqueue('db_writes', {
        'table': 'users',
        'id': user_id,
        'data': data
    })

    return data  # Fast response

# Background worker
def process_db_writes():
    while True:
        write = queue.dequeue('db_writes')
        db.update(...)
```

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

4. Read-Through Cache

Pattern:

```
# Cache layer handles DB queries automatically
user = cache.get_with_loader(
    key=f"user:{user_id}",
    loader=lambda: db.query("SELECT * FROM users WHERE id = ?", user_id),
    ttl=3600
)
```

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

Cache Eviction Policies

1. LRU (Least Recently Used)

```
class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key):
        if key not in self.cache:
            return None
        # Move to end (most recent)
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, value):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = value
        if len(self.cache) > self.capacity:
            # Evict least recently used (first item)
            self.cache.popitem(last=False)
```

2. LFU (Least Frequently Used)

```
class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}  # key -> (value, frequency)
        self.freq_map = defaultdict(list)  # frequency -> [keys]
        self.min_freq = 0

    def get(self, key):
        if key not in self.cache:
            return None
        value, freq = self.cache[key]
```

```
        # Increment frequency
        self.cache[key] = (value, freq + 1)
        self.freq_map[freq].remove(key)
        self.freq_map[freq + 1].append(key)
        return value

    def put(self, key, value):
        if len(self.cache) >= self.capacity:
            # Evict least frequently used
            evict_key = self.freq_map[self.min_freq][0]
            del self.cache[evict_key]
            self.freq_map[self.min_freq].remove(evict_key)

        self.cache[key] = (value, 1)
        self.freq_map[1].append(key)
        self.min_freq = 1
```

3. FIFO (First In First Out)

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

4. TTL (Time To Live)

```
cache.set(key, value, ttl=3600)  # Expire after 1 hour
```

Advanced Caching Techniques

1. Bloom Filters (Negative Cache)

```
# Avoid querying DB for non-existent keys
bloom = BloomFilter(size=1000000, hash_functions=3)

def get_user(user_id):
    # Check bloom filter first
    if not bloom.might_contain(user_id):
        return None  # Definitely doesn't exist

    # Might exist - check cache/DB
    return cache_aside_get(user_id)

def create_user(user_id, data):
    db.insert(...)
    bloom.add(user_id)
```

2. Probabilistic Early Expiration (Thundering Herd Prevention)

```python
import random

def get_with_early_expiration(key, loader, ttl):
    value, expiry = cache.get_with_ttl(key)

    if value is None:
        # Cache miss - load data
        value = loader()
        cache.set(key, value, ttl=ttl)
        return value

    # Calculate time to expiry
    remaining = expiry - time.time()

    # Probabilistic early refresh
    # Higher probability as expiry approaches
    probability = 1 - (remaining / ttl)
    if random.random() < probability:
        # Async refresh
        async_refresh(key, loader, ttl)

    return value
```

3. Consistent Hashing for Cache Distribution

```python
class ConsistentHashRing:
    def __init__(self, nodes, virtual_nodes=150):
        self.ring = {}
        for node in nodes:
            for i in range(virtual_nodes):
                hash_key = hashlib.md5(f"{node}:{i}".encode()).digest()
                self.ring[hash_key] = node
        self.sorted_keys = sorted(self.ring.keys())

    def get_node(self, key):
        if not self.ring:
            return None
        hash_key = hashlib.md5(key.encode()).digest()
        for ring_key in self.sorted_keys:
            if hash_key <= ring_key:
                return self.ring[ring_key]
        return self.ring[self.sorted_keys[0]]

# Usage
cache_nodes = ["cache1:6379", "cache2:6379", "cache3:6379"]
ring = ConsistentHashRing(cache_nodes)

def cache_get(key):
    node = ring.get_node(key)
    return redis.connect(node).get(key)
```

Monitoring Cache Performance

Key Metrics:

```
cache_hit_rate = cache_hits / (cache_hits + cache_misses)
# Target: > 80% for most applications

cache_eviction_rate = evictions / total_operations
# High rate indicates cache too small

average_ttl_hit_rate = hits_before_expiry / total_sets
# Low rate indicates TTL too short

memory_utilization = used_memory / max_memory
# Target: 70-80% (headroom for spikes)
```

---

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:

```python
from abc import ABC, abstractmethod
import csv
import json
import xml.etree.ElementTree as ET

class FileAdapter(ABC):
    @abstractmethod
    def read(self, filepath):
        pass

    @abstractmethod
    def write(self, filepath, data):
        pass

class CSVAdapter(FileAdapter):
    def read(self, filepath):
        with open(filepath, 'r') as file:
            reader = csv.DictReader(file)
            return list(reader)
```

```
    def write(self, filepath, data):
        if not data:
            return
        with open(filepath, 'w', newline='') as file:
            writer = csv.DictWriter(file, fieldnames=data[0].keys())
            writer.writeheader()
            writer.writerows(data)

class JSONAdapter(FileAdapter):
    def read(self, filepath):
        with open(filepath, 'r') as file:
            return json.load(file)

    def write(self, filepath, data):
        with open(filepath, 'w') as file:
            json.dump(data, file, indent=2)

class XMLAdapter(FileAdapter):
    def read(self, filepath):
        tree = ET.parse(filepath)
        root = tree.getroot()
        # Convert XML to dict (simplified)
        return self._xml_to_dict(root)

    def write(self, filepath, data):
        root = self._dict_to_xml(data)
        tree = ET.ElementTree(root)
        tree.write(filepath)

    def _xml_to_dict(self, element):
        # Implementation details...
        pass

# Factory pattern for adapter selection
class FileAdapterFactory:
    @staticmethod
    def get_adapter(file_type):
        adapters = {
            'csv': CSVAdapter(),
            'json': JSONAdapter(),
            'xml': XMLAdapter()
        }
        return adapters.get(file_type.lower())

# Usage
adapter = FileAdapterFactory.get_adapter('csv')
data = adapter.read('data.csv')
processed_data = process(data)
adapter.write('output.csv', processed_data)
```

Advanced File Adapter (Streaming for Large Files):

```python
class StreamingCSVAdapter:
    def read_stream(self, filepath, chunk_size=1000):
        with open(filepath, 'r') as file:
            reader = csv.DictReader(file)
            chunk = []
            for row in reader:
                chunk.append(row)
                if len(chunk) >= chunk_size:
                    yield chunk
                    chunk = []
            if chunk:
                yield chunk

    def write_stream(self, filepath, data_generator):
        first_chunk = next(data_generator)
        with open(filepath, 'w', newline='') as file:
            writer = csv.DictWriter(file,
fieldnames=first_chunk[0].keys())
            writer.writeheader()
            writer.writerows(first_chunk)

            for chunk in data_generator:
                writer.writerows(chunk)

# Usage for large files
adapter = StreamingCSVAdapter()
for chunk in adapter.read_stream('large_file.csv', chunk_size=10000):
    process_chunk(chunk)
```

FTP Adapter

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

Implementation:

```python
from ftplib import FTP, FTP_TLS
import os

class FTPAdapter:
    def __init__(self, host, username, password, port=21, use_tls=False):
        self.host = host
        self.username = username
        self.password = password
        self.port = port
        self.use_tls = use_tls
        self.ftp = None

    def connect(self):
        if self.use_tls:
            self.ftp = FTP_TLS()
```

```python
        else:
            self.ftp = FTP()

        self.ftp.connect(self.host, self.port)
        self.ftp.login(self.username, self.password)

        if self.use_tls:
            self.ftp.prot_p()  # Set up secure data connection

        return self

    def disconnect(self):
        if self.ftp:
            self.ftp.quit()

    def upload(self, local_path, remote_path):
        with open(local_path, 'rb') as file:
            self.ftp.storbinary(f'STOR {remote_path}', file)

    def download(self, remote_path, local_path):
        with open(local_path, 'wb') as file:
            self.ftp.retrbinary(f'RETR {remote_path}', file.write)

    def list_files(self, remote_dir='/'):
        self.ftp.cwd(remote_dir)
        return self.ftp.nlst()

    def delete(self, remote_path):
        self.ftp.delete(remote_path)

    def create_directory(self, remote_dir):
        self.ftp.mkd(remote_dir)

    def __enter__(self):
        return self.connect()

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.disconnect()

# Usage
with FTPAdapter('ftp.example.com', 'user', 'pass', use_tls=True) as ftp:
    # Upload file
    ftp.upload('local_data.csv', '/remote/data.csv')

    # List files
    files = ftp.list_files('/remote')

    # Download file
    ftp.download('/remote/results.csv', 'local_results.csv')
```

Advanced FTP Adapter (Retry, Logging, Progress):

```
import time
import logging
from tqdm import tqdm

class AdvancedFTPAdapter(FTPAdapter):
    def __init__(self, *args, max_retries=3, retry_delay=5, **kwargs):
        super().__init__(*args, **kwargs)
        self.max_retries = max_retries
        self.retry_delay = retry_delay
        self.logger = logging.getLogger(__name__)

    def _retry_operation(self, operation, *args, **kwargs):
        for attempt in range(self.max_retries):
            try:
                return operation(*args, **kwargs)
            except Exception as e:
                self.logger.warning(f"Attempt {attempt + 1} failed: {e}")
                if attempt < self.max_retries - 1:
                    time.sleep(self.retry_delay)
                    # Reconnect
                    self.disconnect()
                    self.connect()
                else:
                    raise

    def upload_with_progress(self, local_path, remote_path):
        file_size = os.path.getsize(local_path)

        with open(local_path, 'rb') as file:
            with tqdm(total=file_size, unit='B', unit_scale=True) as pbar:
                def callback(data):
                    pbar.update(len(data))

                self._retry_operation(
                    self.ftp.storbinary,
                    f'STOR {remote_path}',
                    file,
                    callback=callback
                )

    def sync_directory(self, local_dir, remote_dir):
        """Sync local directory to remote"""
        for root, dirs, files in os.walk(local_dir):
            # Create remote directories
            rel_path = os.path.relpath(root, local_dir)
            if rel_path != '.':
                remote_path = f"{remote_dir}/{rel_path}"
                try:
                    self.create_directory(remote_path)
                except:
                    pass  # Directory might exist

            # Upload files
```

```
            for file in files:
                local_file = os.path.join(root, file)
                remote_file = f"{remote_dir}/{rel_path}/{file}"
                self.logger.info(f"Uploading {local_file} to
{remote_file}")
                self.upload_with_progress(local_file, remote_file)

# Usage
adapter = AdvancedFTPAdapter(
    'ftp.example.com',
    'user',
    'pass',
    use_tls=True,
    max_retries=3
)

with adapter:
    # Sync entire directory
    adapter.sync_directory('/local/data', '/remote/backup')
```

SFTP Adapter (SSH File Transfer)

```
import paramiko

class SFTPAdapter:
    def __init__(self, host, username, password=None, key_file=None,
port=22):
        self.host = host
        self.username = username
        self.password = password
        self.key_file = key_file
        self.port = port
        self.transport = None
        self.sftp = None

    def connect(self):
        self.transport = paramiko.Transport((self.host, self.port))

        if self.key_file:
            private_key =
paramiko.RSAKey.from_private_key_file(self.key_file)
            self.transport.connect(username=self.username,
pkey=private_key)
        else:
            self.transport.connect(username=self.username,
password=self.password)

        self.sftp = paramiko.SFTPClient.from_transport(self.transport)
        return self

    def disconnect(self):
```

```
        if self.sftp:
            self.sftp.close()
        if self.transport:
            self.transport.close()

    def upload(self, local_path, remote_path):
        self.sftp.put(local_path, remote_path)

    def download(self, remote_path, local_path):
        self.sftp.get(remote_path, local_path)

    def list_files(self, remote_dir='/'):
        return self.sftp.listdir(remote_dir)

    def __enter__(self):
        return self.connect()

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.disconnect()
```

Use Cases in System Design

1. ETL Pipelines:

```
# Extract from FTP, Transform, Load to DB
with FTPAdapter('ftp.source.com', 'user', 'pass') as ftp:
    ftp.download('/data/export.csv', 'temp/export.csv')

csv_adapter = CSVAdapter()
data = csv_adapter.read('temp/export.csv')

transformed = transform_data(data)

db.bulk_insert('target_table', transformed)
```

2. Legacy System Integration:

```
# Many legacy systems only support FTP for data exchange
class LegacySystemAdapter:
    def __init__(self):
        self.ftp = FTPAdapter('legacy.ftp.com', 'user', 'pass')

    def export_orders(self, orders):
        # Convert modern format to legacy CSV
        csv_adapter = CSVAdapter()
        csv_adapter.write('orders.csv', orders)

        # Upload to legacy FTP
        with self.ftp:
```

```
            self.ftp.upload('orders.csv', '/import/orders.csv')

    def import_results(self):
        # Download from FTP
        with self.ftp:
            self.ftp.download('/export/results.csv', 'results.csv')

        # Parse and return
        csv_adapter = CSVAdapter()
        return csv_adapter.read('results.csv')
```

---

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:

```
# Bank account transfer (must be strongly consistent)
def transfer(from_account, to_account, amount):
    with db.transaction():  # ACID transaction
        # Read current balances
        from_balance = db.query("SELECT balance FROM accounts WHERE id = 
?", from_account)
        to_balance = db.query("SELECT balance FROM accounts WHERE id = ?", 
to_account)

        # Update balances
        db.execute("UPDATE accounts SET balance = ? WHERE id = ?",
                   from_balance - amount, from_account)
        db.execute("UPDATE accounts SET balance = ? WHERE id = ?",
                   to_balance + amount, to_account)

        # 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:

```python
# Social media likes (eventual consistency acceptable)
def like_post(post_id, user_id):
    # Write to local datacenter (fast)
    local_db.execute("INSERT INTO likes (post_id, user_id) VALUES (?, ?)",
                     post_id, user_id)

    # Asynchronously replicate to other datacenters
    replication_queue.enqueue({
        'operation': 'insert',
        'table': 'likes',
        'data': {'post_id': post_id, 'user_id': user_id}
    })

    # Immediate response to user
    return "Liked!"

# User in another datacenter might not see the 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):

```
class EventuallyConsistentDB:
    def write(self, key, value):
        timestamp = time.time()
        self.store(key, value, timestamp)
        self.replicate_async(key, value, timestamp)

    def merge_conflict(self, local_value, remote_value):
        # Resolve by timestamp (LWW)
        if remote_value.timestamp > local_value.timestamp:
            return remote_value
        return local_value
```

2. Vector Clocks (Detect Conflicts):

```
# Track causality across replicas
vector_clock = {
    'replica_1': 5,  # 5 writes on replica 1
    'replica_2': 3,  # 3 writes on replica 2
    'replica_3': 7   # 7 writes on replica 3
}

# Concurrent writes = conflict
# Application must resolve
```

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

```python
# G-Counter (Grow-only counter)
class GCounter:
    def __init__(self, replica_id):
        self.replica_id = replica_id
        self.counts = defaultdict(int)

    def increment(self):
        self.counts[self.replica_id] += 1

    def value(self):
        return sum(self.counts.values())

    def merge(self, other):
        for replica, count in other.counts.items():
            self.counts[replica] = max(self.counts[replica], count)

# Automatically resolves conflicts without coordination
```

Trade-offs Summary

Aspect	Strong	Eventual
Consistency	Immediate	Delayed
Latency	Higher	Lower
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):

```python
class TwoPhaseCommit:
    def __init__(self, participants):
        self.participants = participants

    def execute_transaction(self, transaction):
```

```
        # Phase 1: Prepare
        prepare_results = []
        for participant in self.participants:
            result = participant.prepare(transaction)
            prepare_results.append(result)

        # 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):

```
class Saga:
    def __init__(self):
        self.steps = []
        self.compensations = []

    def add_step(self, action, compensation):
        self.steps.append(action)
        self.compensations.append(compensation)

    async def execute(self):
```

```
        executed_steps = []
        try:
            for step in self.steps:
                await step()
                executed_steps.append(step)
        except Exception as e:
            # Rollback: Execute compensations in reverse
            for i in range(len(executed_steps) - 1, -1, -1):
                await self.compensations[i]()
            raise

# Example: E-commerce order
saga = Saga()
saga.add_step(
    action=lambda: reserve_inventory(product_id, quantity),
    compensation=lambda: release_inventory(product_id, quantity)
)
saga.add_step(
    action=lambda: charge_payment(user_id, amount),
    compensation=lambda: refund_payment(user_id, amount)
)
saga.add_step(
    action=lambda: create_shipment(order_id),
    compensation=lambda: cancel_shipment(order_id)
)

await saga.execute()
```

Conflict Resolution Strategies

1. Last-Write-Wins (LWW):

```
def resolve_conflict(local_doc, remote_doc):
    if remote_doc.timestamp > local_doc.timestamp:
        return remote_doc
    return local_doc
```

Issue: Can lose concurrent writes

2. Application-Specific Logic:

```
def resolve_shopping_cart(local_cart, remote_cart):
    # Union of items (merge)
    merged_items = {}
    for item in local_cart.items + remote_cart.items:
        if item.id in merged_items:
            # Keep max quantity
            merged_items[item.id].quantity = max(
                merged_items[item.id].quantity,
```

```
                item.quantity
            )
        else:
            merged_items[item.id] = item
    return merged_items.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:

```
# Replication lag
replication_lag = master_timestamp - replica_timestamp
# Alert if > 5 seconds

# Consistency violations
def check_consistency():
    master_count = master_db.count('users')
    replica_count = replica_db.count('users')
    difference = abs(master_count - replica_count)
    # Alert if difference > threshold

# Conflict rate
conflict_rate = conflicts_detected / total_writes
# Monitor for spikes
```

---

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.

```
import time

class TokenBucket:
    def __init__(self, capacity, refill_rate):
        self.capacity = capacity  # Max tokens
        self.tokens = capacity
        self.refill_rate = refill_rate  # Tokens per second
        self.last_refill = time.time()

    def allow_request(self):
        self._refill()
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False

    def _refill(self):
        now = time.time()
        elapsed = now - self.last_refill
        tokens_to_add = elapsed * self.refill_rate
        self.tokens = min(self.capacity, self.tokens + tokens_to_add)
        self.last_refill = now

# Usage
limiter = TokenBucket(capacity=100, refill_rate=10)  # 100 tokens, 10/sec
refill

if limiter.allow_request():
    process_request()
else:
    return "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.

```
from collections import deque
import time

class LeakyBucket:
    def __init__(self, capacity, leak_rate):
        self.capacity = capacity  # Max queue size
        self.leak_rate = leak_rate  # Requests per second
        self.queue = deque()
        self.last_leak = time.time()

    def allow_request(self):
        self._leak()
        if len(self.queue) < self.capacity:
```

```
            self.queue.append(time.time())
            return True
        return False

    def _leak(self):
        now = time.time()
        elapsed = now - self.last_leak
        leaks = int(elapsed * self.leak_rate)

        for _ in range(min(leaks, len(self.queue))):
            self.queue.popleft()

        self.last_leak = now
```

Pros: Smooth output rate, prevents spikes **Cons**: Can queue requests (latency)

3. Fixed Window Counter

```python
import time

class FixedWindowCounter:
    def __init__(self, limit, window_seconds):
        self.limit = limit
        self.window_seconds = window_seconds
        self.count = 0
        self.window_start = time.time()

    def allow_request(self):
        now = time.time()

        # Reset window if expired
        if now - self.window_start >= self.window_seconds:
            self.count = 0
            self.window_start = now

        if self.count < self.limit:
            self.count += 1
            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

```python
import time
from collections import deque

class SlidingWindowLog:
    def __init__(self, limit, window_seconds):
        self.limit = limit
```

```
        self.window_seconds = window_seconds
        self.requests = deque()  # Timestamps

    def allow_request(self):
        now = time.time()

        # Remove expired entries
        cutoff = now - self.window_seconds
        while self.requests and self.requests[0] < cutoff:
            self.requests.popleft()

        if len(self.requests) < self.limit:
            self.requests.append(now)
            return True
        return False
```

Pros: Accurate, no boundary issues **Cons**: Memory grows with request count

5. Sliding Window Counter (Redis)

```
import redis
import time

class SlidingWindowRedis:
    def __init__(self, redis_client, limit, window_seconds):
        self.redis = redis_client
        self.limit = limit
        self.window_seconds = window_seconds

    def allow_request(self, user_id):
        key = f"rate_limit:{user_id}"
        now = time.time()
        window_start = now - self.window_seconds

        # Lua script for atomic operation
        lua_script = """
        local key = KEYS[1]
        local now = tonumber(ARGV[1])
        local window_start = tonumber(ARGV[2])
        local limit = tonumber(ARGV[3])

        -- 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
        """

        result = self.redis.eval(
            lua_script,
            1,
            key,
            now,
            window_start,
            self.limit
        )

        return result == 1
```

Distributed Rate Limiting

Challenge: Multiple API servers need shared rate limit state.

Solution 1: Centralized Counter (Redis):

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

    def check_rate_limit(self, key, limit, window_seconds):
        pipe = self.redis.pipeline()
        now = int(time.time())
        window_key = f"{key}:{now // window_seconds}"

        pipe.incr(window_key)
        pipe.expire(window_key, window_seconds * 2)
        result = pipe.execute()

        count = result[0]
        return count <= limit

# Usage across multiple servers
if not limiter.check_rate_limit(f"user:{user_id}", limit=100,
window_seconds=60):
    return "Rate limit exceeded", 429
```

Solution 2: Distributed Token Bucket:

```
def distributed_token_bucket(user_id, capacity, refill_rate):
    key = f"token_bucket:{user_id}"

    # Lua script for atomic token bucket
    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
    """

    result = redis_client.eval(
        lua_script,
        1,
        key,
        capacity,
        refill_rate,
        time.time()
    )

    return result == 1
```

Tiered Rate Limiting

```
class TieredRateLimiter:
    def __init__(self):
        self.limits = {
            'free': {'requests': 100, 'window': 3600},  # 100/hour
            'basic': {'requests': 1000, 'window': 3600},  # 1000/hour
            'premium': {'requests': 10000, 'window': 3600},  # 10000/hour
        }

    def check_limit(self, user_id, tier):
        config = self.limits.get(tier, self.limits['free'])
        return self.check_rate_limit(
            f"user:{user_id}:{tier}",
            config['requests'],
```

```
            config['window']
        )
```

Rate Limiting by Multiple Dimensions

```python
class MultiDimensionRateLimiter:
    def allow_request(self, user_id, api_key, ip_address):
        # Check multiple limits
        checks = [
            ('user', user_id, 1000, 60),  # 1000/min per user
            ('api_key', api_key, 5000, 60),  # 5000/min per API key
            ('ip', ip_address, 100, 60),  # 100/min per IP
            ('global', 'all', 50000, 60),  # 50000/min globally
        ]

        for dimension, key, limit, window in checks:
            if not self.check_rate_limit(f"{dimension}:{key}", limit,
window):
                return False, f"Rate limit exceeded for {dimension}"

        return True, None
```

Response Headers

```python
def add_rate_limit_headers(response, remaining, limit, reset_time):
    response.headers['X-RateLimit-Limit'] = str(limit)
    response.headers['X-RateLimit-Remaining'] = str(remaining)
    response.headers['X-RateLimit-Reset'] = str(reset_time)

    if remaining == 0:
        response.headers['Retry-After'] = str(int(reset_time -
time.time()))

    return response
```

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 mmh3  # MurmurHash
import numpy as np

class CountMinSketch:
```

```
    def __init__(self, width, depth):
        self.width = width  # Number of counters per row
        self.depth = depth  # Number of hash functions
        self.table = np.zeros((depth, width), dtype=np.int64)

    def _hash(self, item, seed):
        return mmh3.hash(str(item), seed) % self.width

    def add(self, item, count=1):
        for i in range(self.depth):
            index = self._hash(item, i)
            self.table[i][index] += count

    def estimate(self, item):
        # Return minimum count across all rows
        counts = [self.table[i][self._hash(item, i)] for i in
range(self.depth)]
        return min(counts)

# Top K with Min-Heap
class TopKHeavyHitter:
    def __init__(self, k, width=10000, depth=7):
        self.k = k
        self.cms = CountMinSketch(width, depth)
        self.min_heap = []  # (count, item)
        self.items_in_heap = set()

    def add(self, item):
        self.cms.add(item)
        count = self.cms.estimate(item)

        if item in self.items_in_heap:
            # Update existing entry
            self.min_heap = [(c, i) for c, i in self.min_heap if i !=
item]
            heapq.heapify(self.min_heap)
            heapq.heappush(self.min_heap, (count, item))
        elif len(self.min_heap) < self.k:
            # Heap not full
            heapq.heappush(self.min_heap, (count, item))
            self.items_in_heap.add(item)
        elif count > self.min_heap[0][0]:
            # Replace minimum
            _, evicted = heapq.heapreplace(self.min_heap, (count, item))
            self.items_in_heap.remove(evicted)
            self.items_in_heap.add(item)

    def get_top_k(self):
        return sorted(self.min_heap, reverse=True)

# Usage
hh = TopKHeavyHitter(k=100, width=100000, depth=7)
for item in stream:
    hh.add(item)
```

```
top_100 = hh.get_top_k()
```

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

```
class LossyCounting:
    def __init__(self, support_threshold, error=0.001):
        self.support = support_threshold
        self.error = error
        self.bucket_width = int(1 / error)
        self.current_bucket = 1
        self.counts = {}  # item -> (count, delta)
        self.n = 0  # Total items processed

    def add(self, item):
        self.n += 1

        if item in self.counts:
            count, delta = self.counts[item]
            self.counts[item] = (count + 1, delta)
        else:
            self.counts[item] = (1, self.current_bucket - 1)

        # Check if bucket boundary
        if self.n % self.bucket_width == 0:
            self.current_bucket += 1
            self._prune()

    def _prune(self):
        # Remove items with count + delta <= current_bucket
        to_remove = []
        for item, (count, delta) in self.counts.items():
            if count + delta <= self.current_bucket:
                to_remove.append(item)

        for item in to_remove:
            del self.counts[item]

    def get_frequent_items(self):
        threshold = self.support * self.n
        return [(item, count) for item, (count, _) in self.counts.items()
                if count >= threshold]
```

Memory: O(1/ε) where ε = error threshold **Accuracy**: Guarantees: no false negatives, but possible false positives

4. Space-Saving Algorithm

```python
import heapq

class SpaceSaving:
    def __init__(self, k):
        self.k = k
        self.counters = {}  # item -> count
        self.min_heap = []  # (count, item)

    def add(self, item):
        if item in self.counters:
            # Increment existing counter
            self.counters[item] += 1
        elif len(self.counters) < self.k:
            # Add new counter
            self.counters[item] = 1
            heapq.heappush(self.min_heap, (1, item))
        else:
            # Replace minimum counter
            min_count, min_item = heapq.heappop(self.min_heap)
            del self.counters[min_item]
            self.counters[item] = min_count + 1
            heapq.heappush(self.min_heap, (min_count + 1, item))

    def get_top_k(self):
        return sorted(self.counters.items(), key=lambda x: x[1],
reverse=True)
```

Memory: O(k) **Accuracy**: Guarantees top k items within error bound

Distributed Top K

MapReduce Approach:

```python
# Map phase: Each worker maintains local top K
class Mapper:
    def __init__(self, k):
        self.local_topk = TopKHeavyHitter(k)

    def process_chunk(self, data_chunk):
        for item in data_chunk:
            self.local_topk.add(item)
        return self.local_topk.get_top_k()

# Reduce phase: Merge local top K
class Reducer:
    def __init__(self, k):
        self.k = k
        self.global_counts = Counter()

    def merge(self, local_topk_results):
```

```
        for topk_list in local_topk_results:
            for count, item in topk_list:
                self.global_counts[item] += count

        return heapq.nlargest(self.k, self.global_counts.items(),
                              key=lambda x: x[1])

# Usage
mappers = [Mapper(k=100) for _ in range(num_workers)]
reducer = Reducer(k=100)

# Parallel processing
local_results = parallel_map(lambda m, chunk: m.process_chunk(chunk),
                              mappers, data_chunks)

# Merge
global_top_100 = reducer.merge(local_results)
```

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):

```
from pyspark.streaming import StreamingContext

def update_top_k(new_values, state):
    topk = state or TopKHeavyHitter(k=100)
    for value in new_values:
        topk.add(value)
    return topk

# Streaming context
ssc = StreamingContext(spark_context, batch_interval=60)  # 1 min batches
```

```
# Stream from Kafka
logs = ssc.kafkaStream("log-topic")

# Extract URLs from logs
urls = logs.map(lambda log: extract_url(log))

# Maintain state for top K
top_k_state = urls.updateStateByKey(update_top_k)

# Output to Redis every minute
top_k_state.foreachRDD(lambda rdd: rdd.foreach(lambda kv:
    redis.set(f"topk:{kv[0]}", json.dumps(kv[1].get_top_k()))))

ssc.start()
ssc.awaitTermination()
```

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...