URL Shortener System Design - High-Level Design (HLD)

Table of Contents

Problem Statement

Design a URL shortening service like bit.ly or TinyURL that:

- Converts long URLs into short, manageable links
- Redirects short URLs to original long URLs
- Tracks analytics (clicks, geographic data, referrers)
- Provides custom aliases (vanity URLs)
- Expires links after a certain time period

Example

```
Long URL:  https://www.example.com/articles/2025/system-design-
interview-prep?utm_source=twitter&utm_medium=social
Short URL: https://bit.ly/3xK9mZ

When user clicks short URL → Redirect to long URL
```

Scale Requirements

- **100 million URLs** shortened per month
- **10 billion redirects** per month
- **Read-to-write ratio**: 100:1
- **URL lifetime**: 10 years default
- **Latency**: < 100ms for redirects

Functional Requirements

Must Have (P0)

1. **URL Shortening**
    - Generate short URL from long URL
    - Short URL should be as short as possible
    - Unique short URLs (no collisions)
    - Default expiration: 10 years
2. **URL Redirection**
    - Redirect short URL to original long URL
    - Handle invalid/expired URLs gracefully
    - Support HTTP 301 (permanent) and 302 (temporary) redirects
3. **Custom Aliases**
    - Users can specify custom short URLs
    - Check availability
    - Example: bit.ly/my-company instead of bit.ly/a3X9z
4. **Analytics**
    - Track click count
    - Track when URL was accessed
    - Geographic location of clicks
    - Referrer information
    - Device type (mobile, desktop)

Nice to Have (P1)

- URL expiration dates (custom TTL)
- User accounts and URL management dashboard
- QR code generation
- Link preview (title, description, thumbnail)
- Branded domains (custom.domain.com/abc)
- Link editing (change destination)
- Password-protected URLs
- API rate limiting

Non-Functional Requirements

Performance

- **Redirect latency**: < 100ms for p99
- **URL generation**: < 200ms
- **High availability**: 99.99% uptime

Scalability

- Handle 100M URL creations per month (~40 QPS)
- Handle 10B redirects per month (~4K QPS)
- Support billions of URLs in storage

Availability

- **99.99% uptime** (4 nines)
- No single point of failure
- Graceful degradation

Security

- Prevent malicious URLs
- Rate limiting to prevent abuse
- CAPTCHA for suspicious activity
- Blacklist known spam domains

Reliability

- URLs should never break (unless expired)
- Data durability: 99.999999999%
- Handle traffic spikes (viral links)

Capacity Estimation

Traffic Estimates

```
URL Shortening (Writes):
  100M URLs/month
  100M / (30 days × 86400 sec) ≈ 40 QPS
  Peak (3x): ~120 QPS

URL Redirection (Reads):
  10B redirects/month
  10B / (30 days × 86400 sec) ≈ 3,850 QPS
  Peak (3x): ~11,500 QPS

Read-to-Write Ratio: 10B / 100M = 100:1
```

Storage Estimates

```
Assumptions:
- 100M new URLs per month
- Store for 10 years
```

```
- Average long URL length: 200 characters

Total URLs over 10 years:
  100M × 12 months × 10 years = 12 billion URLs

Storage per URL:
  Long URL: 200 bytes
  Short code: 7 bytes
  Created timestamp: 8 bytes
  User ID: 8 bytes
  Total: ~225 bytes per URL

Total Storage:
  12B URLs × 225 bytes = 2.7 TB

With analytics (clicks, location, etc.):
  Additional 50 bytes per click
  Assume 10 clicks per URL average
  12B URLs × 10 clicks × 50 bytes = 6 TB

Total: ~9 TB for 10 years
```

Bandwidth Estimates

```
Writes (URL shortening):
  40 QPS × 225 bytes = 9 KB/s

Reads (redirects):
  3,850 QPS × 225 bytes = 866 KB/s
  (Most of this will be served from cache)

Total bandwidth: < 1 MB/s (very manageable)
```

Memory Estimates (Caching)

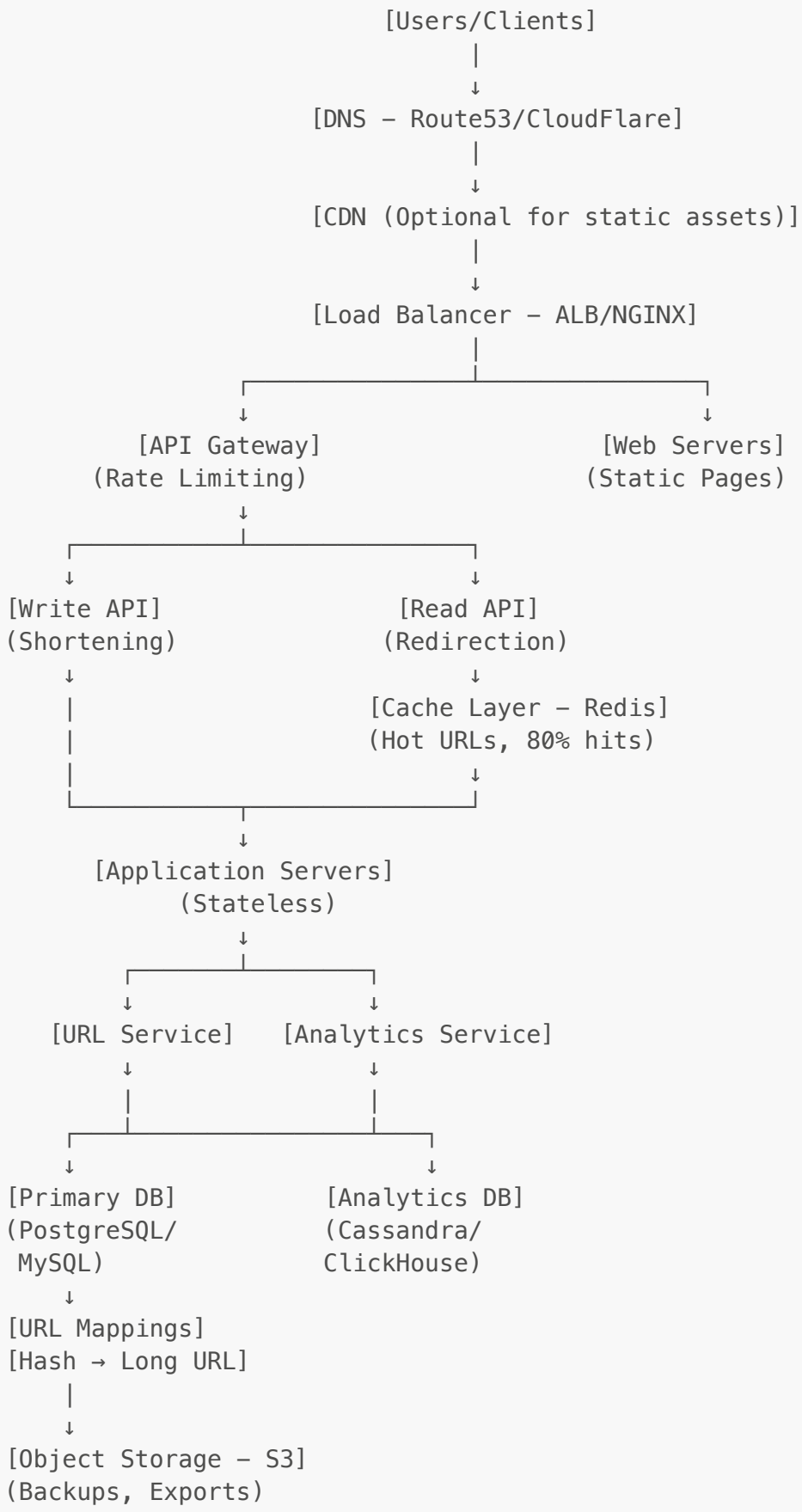
```
80-20 rule: 20% of URLs generate 80% of traffic

Cache 20% of 12B URLs:
  2.4B URLs × 225 bytes = 540 GB

Distributed across 10 cache servers:
  540 GB / 10 = 54 GB per server

Add some buffer for hot URLs:
  ~100 GB per cache server
```

High-Level Architecture



Core Components

1. URL Shortening Service

Responsibilities:

- Generate unique short codes
- Validate long URLs
- Check custom alias availability
- Store URL mappings
- Handle expiration

Key Challenge: Generate unique, short codes efficiently

Solutions (detailed in Deep Dives):

1. Base62 encoding with counter
2. MD5 hash with collision handling
3. Random generation with retry

2. URL Redirection Service

Responsibilities:

- Lookup short code in database/cache
- Return original URL (HTTP 301/302)
- Handle expired/invalid URLs
- Log analytics data asynchronously

Optimization:

- Cache hot URLs in Redis (80%+ cache hit ratio)
- Use CDN for static redirect pages
- Async analytics logging (don't slow down redirect)

3. Analytics Service

Responsibilities:

- Track clicks in real-time
- Store click metadata (timestamp, location, device)
- Aggregate statistics (daily/monthly counts)
- Generate reports

Implementation:

- Write to message queue (Kafka) during redirect
- Batch process into analytics database
- Pre-compute aggregations for dashboards

4. Cache Layer (Redis)

Purpose: Reduce database load, improve redirect latency

What to Cache:

```
Short code → Long URL mapping (80% of traffic)
  Key: "url:{short_code}"
  Value: {long_url, expiration, created_at}
  TTL: 24 hours

Analytics counters:
  Key: "clicks:{short_code}"
  Value: Click count
  TTL: Never (permanent counter)

Rate limiting:
  Key: "rate:{user_ip}:create"
  Value: Request count
  TTL: 1 hour
```

Cache Strategy: Cache-aside (lazy loading)

5. Database (PostgreSQL/MySQL)

Purpose: Source of truth for URL mappings

Why SQL?

- ACID transactions (no lost URLs)
- Simple schema (id, short_code, long_url, created_at)
- Well-understood scaling patterns
- Most URLs fit in single DB

Schema:

```
CREATE TABLE urls (
    id BIGSERIAL PRIMARY KEY,
    short_code VARCHAR(10) UNIQUE NOT NULL,
    long_url VARCHAR(2048) NOT NULL,
    user_id BIGINT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP,
    is_custom BOOLEAN DEFAULT FALSE,
    INDEX idx_short_code (short_code),
    INDEX idx_user_id (user_id),
    INDEX idx_created_at (created_at)
);

CREATE TABLE clicks (
    id BIGSERIAL PRIMARY KEY,
```

```
    short_code VARCHAR(10) NOT NULL,
    clicked_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ip_address VARCHAR(45),
    user_agent TEXT,
    referrer VARCHAR(2048),
    country VARCHAR(2),
    city VARCHAR(100),
    INDEX idx_short_code_time (short_code, clicked_at)
) PARTITION BY RANGE (clicked_at);
```

6. Analytics Database (Cassandra/ClickHouse)

Purpose: Store and analyze billions of click events

Why Cassandra?

- Write-heavy (billions of clicks)
- Time-series data
- Can handle petabytes
- Geographic distribution

Schema:

```
CREATE TABLE click_events (
    short_code text,
    clicked_at timestamp,
    click_id timeuuid,
    ip_address text,
    country text,
    city text,
    device_type text,
    referrer text,
    PRIMARY KEY (short_code, clicked_at, click_id)
) WITH CLUSTERING ORDER BY (clicked_at DESC, click_id DESC);

-- Efficient queries:
-- 1. Get all clicks for a URL
-- 2. Get clicks in time range
-- 3. Sorted by time automatically
```

API Design

RESTful API Endpoints

URL Shortening

```
POST /api/v1/shorten
Request Body:
{
  "long_url": "https://www.example.com/very/long/url",
  "custom_alias": "my-link",  // Optional
  "expiration_date": "2025-12-31T23:59:59Z"  // Optional
}

Response:
{
  "short_url": "https://short.link/a3X9z",
  "short_code": "a3X9z",
  "long_url": "https://www.example.com/very/long/url",
  "created_at": "2025-01-08T14:00:00Z",
  "expires_at": "2035-01-08T14:00:00Z"
}
```

URL Redirection

```
GET /{short_code}

Response: 301/302 Redirect
Location: {long_url}

Or if expired/invalid:
404 Not Found
{
  "error": "URL not found or expired",
  "code": "URL_NOT_FOUND"
}
```

Analytics

```
GET /api/v1/analytics/{short_code}

Response:
{
  "short_code": "a3X9z",
  "total_clicks": 1523,
  "clicks_by_date": [
    {"date": "2025-01-08", "count": 342},
    {"date": "2025-01-07", "count": 581}
  ],
  "clicks_by_country": [
    {"country": "US", "count": 812},
    {"country": "UK", "count": 231}
  ],
```

```
  "clicks_by_device": [
    {"device": "mobile", "count": 923},
    {"device": "desktop", "count": 600}
  ],
  "top_referrers": [
    {"referrer": "twitter.com", "count": 450},
    {"referrer": "facebook.com", "count": 380}
  ]
}
```

Custom Alias Availability

```
GET /api/v1/available/{custom_alias}

Response:
{
  "alias": "my-link",
  "available": true
}
```

Deep Dives

1. Short Code Generation Algorithms

This is THE critical design decision. Three main approaches:

Approach 1: Base62 Encoding (Recommended)

How it works:

```
1. Use auto-incrementing counter in database
2. Encode counter to Base62 (0-9, a-z, A-Z)
3. 62 characters gives us huge space

Math:
- 6 characters: 62^6 = 56 billion combinations
- 7 characters: 62^7 = 3.5 trillion combinations

Example:
Counter: 125
Base62: "cb" (125 / 62 = 2 remainder 1 → "cb")
```

Implementation:

```
BASE62 =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

def encode_base62(number):
    if number == 0:
        return BASE62[0]

    result = []
    while number > 0:
        remainder = number % 62
        result.append(BASE62[remainder])
        number = number // 62

    return ''.join(reversed(result))

def decode_base62(short_code):
    number = 0
    for char in short_code:
        number = number * 62 + BASE62.index(char)
    return number

# Example:
encode_base62(125) → "cb"
decode_base62("cb") → 125
```

Pros:

- Guaranteed unique (counter is unique)
- Short URLs (6-7 characters)
- Predictable length
- No collisions

Cons:

- Sequential (predictable next URL)
- Counter becomes bottleneck
- Single point of failure

Solution for Counter Bottleneck:

- Use distributed counter (Zookeeper, Redis)
- Pre-allocate ranges to each server
  - Server 1: 1-1M
  - Server 2: 1M-2M
  - Server 3: 2M-3M

Approach 2: MD5 Hash (Simple but Collision-Prone)

How it works:

```
1. Hash long URL with MD5
2. Take first 6-8 characters
3. Check for collision
4. If collision, append counter and re-hash

MD5("https://example.com") = "a3c4b5..."
Short code = "a3c4b5" (first 6 chars)
```

Implementation:

```python
import hashlib

def generate_short_code_md5(long_url, length=6):
    hash_value = hashlib.md5(long_url.encode()).hexdigest()
    short_code = hash_value[:length]

    # Check collision
    attempts = 0
    while url_exists(short_code) and attempts < 5:
        # Append counter and re-hash
        hash_value = hashlib.md5(f"{long_url}
{attempts}".encode()).hexdigest()
        short_code = hash_value[:length]
        attempts += 1

    return short_code
```

Pros:

- Same long URL always gets same short code
- Distributed (no central counter)
- Simple implementation

Cons:

- Collision probability increases with scale
- Non-sequential lookups (harder to cache)
- Need collision handling logic

Collision Probability:

```
For 6 characters from MD5 (62^6 = 56B combinations):
After 1M URLs: ~0.001% collision chance
After 100M URLs: ~0.1% collision chance
After 1B URLs: ~10% collision chance
```

Approach 3: Random Generation + Retry

How it works:

```
1. Generate random 6-7 character string
2. Check if exists in database
3. If exists, retry (up to 5 times)
4. If still collision, increase length
```

Implementation:

```python
import random
import string

def generate_short_code_random(length=6, max_retries=5):
    chars = string.ascii_letters + string.digits  # a-z, A-Z, 0-9

    for attempt in range(max_retries):
        short_code = ''.join(random.choices(chars, k=length))

        if not url_exists(short_code):
            return short_code

    # If all retries failed, increase length
    return generate_short_code_random(length + 1, max_retries)
```

Pros:

- Simple implementation
- Non-sequential (harder to guess)
- Distributed (no coordination needed)

Cons:

- Collisions increase with scale
- Need retry logic
- Database check on every generation

Bit.ly's Approach: Random generation with pre-allocated pools

Recommendation: Base62 with Distributed Counter

For interviews, use **Base62 with distributed counter**:

- Mention Zookeeper for coordination
- Or pre-allocate ranges to servers
- Best balance of uniqueness and performance

---

2. URL Redirection Flow

Two Types of Redirects:

HTTP 301 (Permanent Redirect):

- Browser caches the redirect
- Future requests go directly to long URL
- Pros: Faster for user (no server hit)
- Cons: Can't track analytics after first click

HTTP 302 (Temporary Redirect):

- Browser doesn't cache
- Every click goes through our server
- Pros: Can track all clicks
- Cons: Slight latency on every click

Recommendation: Use **302** for analytics, switch to **301** for performance if analytics not needed

Optimized Redirect Flow:

```
1. User clicks: https://short.link/a3X9z

2. Browser → Load Balancer → Application Server

3. Application Server:
   a. Check Redis cache
   b. If cache hit (80%+ of time):
      - Return redirect instantly (< 10ms)
   c. If cache miss:
      - Query database
      - Add to cache (TTL: 24 hours)
      - Return redirect (< 50ms)

4. Async (non-blocking):
   - Publish click event to Kafka
   - Analytics service processes later

5. Browser redirects to long URL
```

Code:

```python
@app.route('/<short_code>')
def redirect_url(short_code):
    # Check cache first
    cached = redis.get(f"url:{short_code}")
    if cached:
        long_url = cached
```

```
    else:
        # Cache miss - query database
        url_obj = db.query("SELECT long_url, expires_at FROM urls WHERE
short_code = ?", short_code)

        if not url_obj:
            return "URL not found", 404

        if url_obj.expires_at < datetime.now():
            return "URL expired", 410

        long_url = url_obj.long_url

        # Cache for 24 hours
        redis.setex(f"url:{short_code}", 86400, long_url)

    # Async analytics (don't block redirect)
    kafka.produce("click_events", {
        "short_code": short_code,
        "timestamp": time.time(),
        "ip": request.remote_addr,
        "user_agent": request.user_agent,
        "referrer": request.referrer
    })

    # Redirect
    return redirect(long_url, code=302)
```

3. Custom Aliases (Vanity URLs)

Challenge: How to handle both generated and custom short codes?

Solution:

```
CREATE TABLE urls (
    id BIGSERIAL PRIMARY KEY,
    short_code VARCHAR(10) UNIQUE NOT NULL,
    long_url VARCHAR(2048) NOT NULL,
    is_custom BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP,
    INDEX idx_short_code (short_code)
);

-- When creating URL:
-- 1. Check if custom_alias provided
-- 2. If yes: Check availability, use if available
-- 3. If no: Generate using Base62
```

Validation for Custom Aliases:

```python
def validate_custom_alias(alias):
    # Length: 4-20 characters
    if len(alias) < 4 or len(alias) > 20:
        return False

    # Only alphanumeric and hyphen
    if not re.match(r'^[a-zA-Z0-9-]+$', alias):
        return False

    # Not reserved words
    reserved = ['api', 'admin', 'dashboard', 'analytics']
    if alias.lower() in reserved:
        return False

    # Check availability
    if url_exists(alias):
        return False

    return True
```

4. Rate Limiting

Why Needed:

- Prevent spam
- Prevent abuse (generating millions of URLs)
- Protect infrastructure

Implementation: Token Bucket in Redis

```python
def check_rate_limit(ip_address, action='create'):
    key = f"rate:{ip_address}:{action}"

    # Limits:
    # - Create URL: 10/hour per IP
    # - Redirect: 1000/hour per IP

    limit = 10 if action == 'create' else 1000
    window = 3600  # 1 hour

    current = redis.get(key)
    if current is None:
        redis.setex(key, window, limit - 1)
        return True

    if int(current) > 0:
        redis.decr(key)
        return True
```

```
    return False  # Rate limited
```

Rate Limits:

```
Anonymous Users:
- 10 URLs per hour
- 1000 redirects per hour

Authenticated Users:
- 100 URLs per hour
- 10,000 redirects per hour

Premium Users:
- 10,000 URLs per hour
- Unlimited redirects
```

5. URL Cleanup (Expired URLs)

Challenge: Delete expired URLs to save space

Solution: Lazy deletion + periodic cleanup

Lazy Deletion:

```
# During redirect, check expiration
if url.expires_at < datetime.now():
    # Don't delete immediately (expensive)
    # Just return 410 Gone
    return "URL expired", 410
```

Periodic Cleanup (daily cron job):

```
-- Run once per day
DELETE FROM urls
WHERE expires_at < NOW()
AND expires_at IS NOT NULL
LIMIT 100000;

-- Do in batches to avoid long-running transactions
```

Alternative: Move to separate "expired_urls" table for analytics

---

Scalability & Reliability

Horizontal Scaling

Application Servers:

- Stateless (can add/remove freely)
- Auto-scaling based on QPS
- Target: 1000 QPS per server
- For 12K peak QPS: 15 servers + 20% buffer = 18 servers

Database Scaling:

```
Phase 1 (0-1B URLs):
- Single PostgreSQL instance
- 5-10 read replicas for redirects

Phase 2 (1B-10B URLs):
- Shard by short_code ranges
- Shard 1: a-g
- Shard 2: h-n
- Shard 3: o-u
- Shard 4: v-9

Phase 3 (>10B URLs):
- Move to DynamoDB or Cassandra
- Infinite horizontal scaling
```

Cache Scaling:

- Redis Cluster with 10 nodes
- Automatic sharding by key
- 3 replicas per shard for HA

High Availability

Multi-Region Setup:

```
Primary Region: us-east-1 (50% traffic)
Secondary Region: us-west-2 (30% traffic)
Tertiary Region: eu-west-1 (20% traffic)

Benefits:
- Low latency globally
- Disaster recovery
- Load distribution
```

Redundancy:

- Multiple load balancers (active-active)

- Database replication (master-slave)
- Cache replication (Redis Sentinel)
- Message queue replication (Kafka clusters)

Failover:

- Automatic health checks every 10s
- Failover time: < 30 seconds
- Use Route53 for DNS failover

Monitoring

Metrics to Track:

```
Application:
- URL creation rate (QPS)
- Redirect latency (p50, p95, p99)
- Cache hit ratio (target: 80%+)
- Error rate (target: < 0.1%)

Infrastructure:
- Server CPU, memory
- Database connection pool
- Cache memory usage

Business:
- Total URLs created
- Active URLs (not expired)
- Total redirects
- Most popular URLs
```

Alerting:

- Latency > 200ms for 5 minutes
- Error rate > 1%
- Cache hit ratio < 70%
- Database connection pool > 80%

Security

URL Validation

Prevent Malicious URLs:

```python
def validate_url(long_url):
    # Parse URL
    parsed = urlparse(long_url)
```

```
    # Must have scheme (http/https)
    if parsed.scheme not in ['http', 'https']:
        return False

    # Must have domain
    if not parsed.netloc:
        return False

    # Check against blacklist
    if is_blacklisted_domain(parsed.netloc):
        return False

    # Check with Google Safe Browsing API
    if is_malicious(long_url):
        return False

    return True
```

Blacklist Domains:

- Maintain list of known phishing/spam domains
- Update regularly from security feeds
- Redis set for fast lookup

Rate Limiting

Why Multiple Layers:

```
Layer 1: CDN/WAF (CloudFlare)
- Block obvious attacks
- DDOS protection

Layer 2: API Gateway
- Per-IP rate limiting
- Per-user rate limiting

Layer 3: Application
- Business logic limits
- Custom rules
```

CAPTCHA

When to Show:

- Anonymous users creating > 5 URLs/hour
- Suspected bot traffic
- After rate limit exceeded

Implementation:

- Google reCAPTCHA v3 (invisible)
- Score < 0.5 → Show challenge
- Integrate with create URL endpoint

Trade-offs & Alternatives

1. Base62 vs MD5 Hash

Chose: Base62 with Distributed Counter

Why:

- Guaranteed unique (no collisions)
- Shorter codes (6-7 chars)
- Predictable length

Trade-off:

- Need distributed counter (Zookeeper complexity)
- Sequential codes (could be guessed)

Alternative: MD5 Hash

- Simpler (no counter needed)
- But: collision handling required
- Bit.ly uses this approach

2. SQL vs NoSQL

Chose: PostgreSQL (start), Cassandra (scale)

Phase 1 (< 1B URLs): PostgreSQL

- Simple, ACID, sufficient for most use cases
- Single instance can handle 10K QPS

Phase 2 (> 1B URLs): Migrate to Cassandra

- Better write performance
- Linear scalability
- No sharding complexity

Trade-off:

- PostgreSQL easier to start
- Cassandra better at scale
- Migration cost

3. 301 vs 302 Redirect

Chose: 302 (Temporary)

Why:

- Can track every click
- Can change destination URL
- Can A/B test

Trade-off:

- Extra server hit on every click
- Slightly higher latency

Alternative: 301

- Better performance (cached)
- But: lose analytics

4. Sync vs Async Analytics

Chose: Async (Kafka)

Why:

- Don't slow down redirects
- Handle traffic spikes
- Can retry failed writes

Implementation:

```
Redirect Request → Return 302 immediately (50ms)
                    ↓
              Kafka Event
                    ↓
      Analytics Worker (processes in batch)
                    ↓
      Cassandra (stores click data)
```

Advanced Features

1. Shortened URL Expiration

Implementation:

```sql
-- Add expiration column
ALTER TABLE urls ADD COLUMN expires_at TIMESTAMP;

-- Index for cleanup query
```

```
CREATE INDEX idx_expires ON urls(expires_at)
WHERE expires_at IS NOT NULL;

-- Check during redirect
SELECT long_url, expires_at
FROM urls
WHERE short_code = ?
AND (expires_at IS NULL OR expires_at > NOW());
```

Cleanup Job (runs daily):

```
# Soft delete (move to archive)
def cleanup_expired_urls():
    batch_size = 10000
    while True:
        expired = db.query("""
            SELECT id, short_code FROM urls
            WHERE expires_at < NOW()
            LIMIT ?
        """, batch_size)

        if not expired:
            break

        # Archive for analytics
        db.insert("archived_urls", expired)

        # Delete from active table
        db.delete("urls", [row.id for row in expired])

        time.sleep(1)  # Don't overload DB
```

2. Custom Branded Domains

Feature: custom.company.com/promo instead of bit.ly/a3X9z

Implementation:

```
1. User registers custom domain
2. Add CNAME record: custom.company.com → short.link
3. Store domain mapping in database

CREATE TABLE custom_domains (
    domain VARCHAR(255) PRIMARY KEY,
    user_id BIGINT NOT NULL,
    verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP
);
```

```
-- During redirect:
-- 1. Extract domain from request
-- 2. Look up owner
-- 3. Find URL mapping
-- 4. Redirect
```

3. Link Preview (Open Graph)

Feature: When shared on social media, show title/image

Implementation:

```
def generate_link_preview(long_url):
    # Fetch URL content
    response = requests.get(long_url, timeout=5)
    soup = BeautifulSoup(response.content, 'html.parser')

    # Extract Open Graph tags
    preview = {
        'title': soup.find('meta', property='og:title'),
        'description': soup.find('meta', property='og:description'),
        'image': soup.find('meta', property='og:image'),
    }

    # Store in database
    db.insert('link_previews', {
        'short_code': short_code,
        'title': preview['title'],
        'description': preview['description'],
        'image_url': preview['image']
    })

    return preview
```

4. QR Code Generation

Feature: Generate QR code for shortened URL

Implementation:

```
import qrcode

def generate_qr_code(short_url):
    qr = qrcode.QRCode(version=1, box_size=10, border=5)
    qr.add_data(short_url)
    qr.make(fit=True)
```

```
    img = qr.make_image(fill_color="black", back_color="white")

    # Save to S3
    s3_key = f"qr/{short_code}.png"
    s3.upload(img, s3_key)

    return f"https://cdn.short.link/{s3_key}"
```

Detailed Component Interactions

URL Creation Flow

```
1. User enters long URL in web interface
2. Frontend calls POST /api/v1/shorten
3. API Gateway:
   - Validates JWT token (if authenticated)
   - Checks rate limit
4. URL Service:
   - Validates URL format
   - Checks URL safety (Google Safe Browsing)
   - Generates short code (Base62)
   - Stores in PostgreSQL
5. Return short URL to user (< 200ms)

Background Processing:
6. Generate QR code (if premium user)
7. Fetch link preview metadata
8. Update search index (Elasticsearch)
```

URL Redirect Flow (Optimized)

```
1. User clicks: https://short.link/a3X9z
2. DNS resolves to nearest datacenter
3. Load Balancer → Application Server
4. Check Redis cache:
   - 80% cache hit → Return in 10ms
   - 20% cache miss → Query DB → Cache → Return in 50ms
5. Log click event to Kafka (async, non-blocking)
6. Return HTTP 302 with Location header
7. Browser redirects to original URL

Analytics Processing (Async):
8. Kafka consumer reads click events
9. Batch write to Cassandra (1000 events at a time)
10. Update counters in Redis
11. Pre-compute daily/monthly aggregates
```

Performance Optimizations

1. Caching Strategy

Cache-Aside Pattern:

```
def get_long_url(short_code):
    # Check cache
    cached = cache.get(f"url:{short_code}")
    if cached:
        return cached

    # Cache miss - query database
    url = db.get(short_code)
    if url:
        # Cache for 24 hours
        cache.set(f"url:{short_code}", url, ttl=86400)

    return url
```

Cache Warming:

- Pre-load top 1000 most popular URLs on startup
- Use analytics data to identify hot URLs
- Schedule cache refresh for trending URLs

Cache Eviction:

- LRU (Least Recently Used) policy
- TTL-based expiration
- Manual purge for updated URLs

2. Database Indexing

Critical Indexes:

```
-- Primary lookup (most important!)
CREATE UNIQUE INDEX idx_short_code ON urls(short_code);

-- User's URLs
CREATE INDEX idx_user_id_created ON urls(user_id, created_at DESC);

-- Cleanup query
CREATE INDEX idx_expires ON urls(expires_at)
WHERE expires_at IS NOT NULL;

-- Analytics queries
```

```
CREATE INDEX idx_clicks_short_code ON clicks(short_code, clicked_at
DESC);
```

Query Performance:

- With index on short_code: < 1ms lookup
- Without index: Full table scan (seconds for billions of rows)

3. Connection Pooling

Database Connections:

```
# Bad: Create new connection per request
def get_url(short_code):
    conn = psycopg2.connect(...)  # Expensive!
    result = conn.execute("SELECT...")
    conn.close()
    return result

# Good: Use connection pool
pool = ConnectionPool(min=10, max=50)

def get_url(short_code):
    with pool.get_connection() as conn:
        result = conn.execute("SELECT...")
    return result
```

Benefits:

- Reuse connections (avoid TCP handshake)
- Limit concurrent connections
- Handle connection failures gracefully

4. Batch Analytics Processing

Don't Process Clicks One-by-One:

```
# Bad: Write to DB on every click (slow!)
def log_click(short_code, metadata):
    db.insert('clicks', {
        'short_code': short_code,
        'clicked_at': datetime.now(),
        ...metadata
    })

# Good: Batch via Kafka
def log_click(short_code, metadata):
    kafka.produce('click_events', {
```

```
        'short_code': short_code,
        ...metadata
    })

# Consumer processes in batches
def consume_click_events():
    batch = []
    for event in kafka.consume(batch_size=1000):
        batch.append(event)

    # Single batch insert
    cassandra.batch_insert('clicks', batch)
```

Benefits:

- 100x faster writes
- Don't block redirects
- Handle traffic spikes

Failure Scenarios & Mitigation

Scenario 1: Database Failure

Impact: Can't create new URLs, can't redirect (if cache miss)
Mitigation:

- Master-slave replication (auto-failover)
- Serve from cache during outage (80% success rate)
- Queue URL creations in Kafka
- RTO: < 1 minute

Scenario 2: Redis Cache Failure

Impact: Increased database load, higher latency
Mitigation:

- Redis Sentinel for automatic failover
- Fall back to database queries
- Rate limit during cache failure
- Impact: 5x slower redirects, but still functional

Scenario 3: Counter Service Failure (Zookeeper)

Impact: Can't generate new short URLs
Mitigation:

- Use pre-allocated ranges (servers have buffer)
- Fall back to random generation temporarily
- Alert ops team

- Queue URL creations until recovery

Scenario 4: Viral Link (Traffic Spike)

Impact: Single URL getting millions of requests
Mitigation:

- CDN caching (redirect page itself)
- Redis cache handles spike
- Auto-scaling adds servers
- No impact if properly architected!

Technology Stack

Layer	Technology	Purpose
DNS	Route53, CloudFlare	Geographic routing
CDN	CloudFront	Static assets, caching
Load Balancer	AWS ALB, NGINX	Traffic distribution
API Gateway	Kong, AWS API Gateway	Rate limiting, auth
App Servers	Node.js, Python (Flask), Go	Business logic
Cache	Redis Cluster	Hot URL lookups
Primary DB	PostgreSQL, MySQL	URL mappings
Analytics DB	Cassandra, ClickHouse	Click events
Message Queue	Apache Kafka	Async processing
Object Storage	Amazon S3	Backups, QR codes
Monitoring	Prometheus, Datadog	Metrics & alerts
Logging	ELK Stack	Centralized logs

Real-World Architectures

Bit.ly Architecture

Stack:

- **MongoDB**: URL storage (flexible schema)
- **Redis**: Caching layer
- **Node.js**: API servers
- **HAProxy**: Load balancing
- **NSQ**: Message queue for analytics

Scale:

- Billions of links
- Billions of clicks per month
- < 10ms redirect latency

TinyURL Architecture

Stack:

- **MySQL**: URL storage
- **Memcached**: Caching
- **PHP**: Application servers
- **Apache**: Web servers

Simple but Effective:

- Handles millions of redirects/day
- Single datacenter initially
- Expanded to multi-region later

Interview Talking Points

Key Design Decisions

1. **Why Base62 encoding?**
    - Guaranteed uniqueness (counter-based)
    - Short codes (6-7 characters)
    - Better than MD5 (no collisions)
2. **Why PostgreSQL initially?**
    - Simple schema, ACID guarantees
    - Handles first 1B URLs easily
    - Can migrate to Cassandra later
3. **Why Redis cache?**
    - 80%+ cache hit ratio
    - Reduces DB load by 80%
    - Sub-millisecond lookups
4. **Why 302 redirect?**
    - Track all clicks for analytics
    - Can change destination URL
    - Trade-off: Extra server hit
5. **Why async analytics?**

- Don't slow down redirects
- Handle traffic spikes
- Better user experience

Potential Bottlenecks & Solutions

1. **Counter becomes bottleneck** → Distribute counter (Zookeeper) or pre-allocate ranges
2. **Database writes slow** → Batch inserts, use write-optimized DB (Cassandra)
3. **Cache memory full** → LRU eviction, scale horizontally
4. **Viral link overwhelms system** → CDN caching, Redis handles spike
5. **Analytics slow down redirects** → Async processing via Kafka

---

Cost Analysis

Infrastructure Costs (Monthly)

For 100M URLs, 10B redirects/month:

```
Application Servers: 20 instances
- EC2 t3.large × 20 = $1,200

Load Balancers: 2 ALBs
- $50/month

Database (PostgreSQL RDS):
- db.r5.2xlarge = $700
- Read replicas × 5 = $3,500

Cache (Redis Cluster):
- 10 nodes × cache.r5.large = $1,500

Analytics (Cassandra):
- 10 nodes × i3.2xlarge = $4,000

S3 Storage (9TB):
- $230

CloudFront CDN:
- $500 (with high cache hit ratio)

Kafka:
- 3 brokers × $300 = $900

Monitoring (Datadog):
- $500

Total: ~$13,000/month

Revenue potential: 10B redirects × $0.001 CPM = $10,000/month
```

```
advertising
(Need premium features for profitability)
```

Interview Answer Template

When asked "Design a URL shortener":

Step 1: Clarify Requirements (2 minutes)

```
Questions to ask:
- How many URLs shortened per day? (100M/month)
- How many redirects per day? (10B/month)
- How long to store URLs? (10 years default)
- Need analytics? (Yes)
- Custom aliases? (Yes)
- QPS? (40 write, 4K read)
```

Step 2: High-Level Design (3 minutes)

```
Components:
1. API Gateway (rate limiting)
2. Application servers (stateless)
3. PostgreSQL (URL storage)
4. Redis (caching)
5. Cassandra (analytics)
6. Kafka (async processing)
```

Step 3: Deep Dive (10 minutes)

```
Focus on:
1. Short code generation (Base62 algorithm)
2. Caching strategy (cache-aside, 80% hit ratio)
3. Database schema
4. Scaling strategy (sharding, replication)
```

Step 4: Bottlenecks & Scaling (5 minutes)

```
Bottlenecks:
1. Database → Add read replicas, then shard
2. Counter → Distribute with Zookeeper
3. Cache → Redis Cluster
```

```
Mention numbers:
- "At 4K QPS, single PostgreSQL handles traffic"
- "With 80% cache hit, only 800 QPS hit database"
- "Read replicas scale to 10K+ QPS"
```

References & Further Reading

System Design Resources

1. **System Design Primer** - GitHub (donnemartin)
2. **Grokking System Design Interview** - educative.io
3. **System Design Interview** - Alex Xu (Book)

Relevant Papers & Articles

1. **Base62 Encoding** - Wikipedia
2. **Consistent Hashing** - MIT Paper
3. **Redis Architecture** - Redis Labs
4. **Cassandra Architecture** - DataStax

Real-World Examples

1. **Bit.ly Engineering Blog**
2. **TinyURL Architecture**
3. **Google URL Shortener (now defunct)**

Appendix

Base62 Encoding Reference

```
Characters: 0-9 (10), a-z (26), A-Z (26) = 62 total

Length    Combinations      URLs/day to exhaust
6 chars   56 billion       153,000 years
7 chars   3.5 trillion     9.5 million years

For 100M URLs/month:
- 6 characters sufficient for 46 years
- 7 characters sufficient for 2,900 years
```

Latency Numbers

```
Redis cache hit:       < 1 ms
Redis cache miss:      1-5 ms
```

```
PostgreSQL query:      1-10 ms
HTTP redirect:         < 100 ms (target)
Analytics write:       N/A (async)
```

QPS Scaling

```
Component             Max QPS per Instance
API Server           1,000-2,000
PostgreSQL (read)    10,000-50,000
PostgreSQL (write)   1,000-10,000
Redis                100,000-1,000,000
Cassandra (write)    10,000+ per node
```

Storage Growth

```
Month 1:    100M URLs × 225 bytes = 22.5 GB
Year 1:     1.2B URLs × 225 bytes = 270 GB
Year 5:     6B URLs × 225 bytes = 1.35 TB
Year 10:    12B URLs × 225 bytes = 2.7 TB

With analytics (10 clicks per URL avg):
Year 10:    12B × 10 × 50 bytes = 6 TB

Total after 10 years: ~9 TB
```

Document Version: 1.0
Last Updated: January 8, 2025
Author: System Design Interview Prep
Status: Complete & Interview-Ready ✅

Pro Tip: This is one of the most common system design questions. Master this, and you'll impress any interviewer!