# Chapter 22: Location-Based Services

## Introduction: The Location Challenge

Finding nearby places/users efficiently at scale.

Naive Approach (Check all):
Find restaurants within 5km of user

SELECT * FROM restaurants
WHERE distance(lat, lng, user_lat, user_lng) < 5;

For 1 million restaurants:
- Calculate 1 million distances
- Sort by distance
- Time: 5+ seconds ✗

Optimized Approach (Geospatial Index):
Use QuadTree or Geohash

Time: 50ms for same query ✓
100x faster!

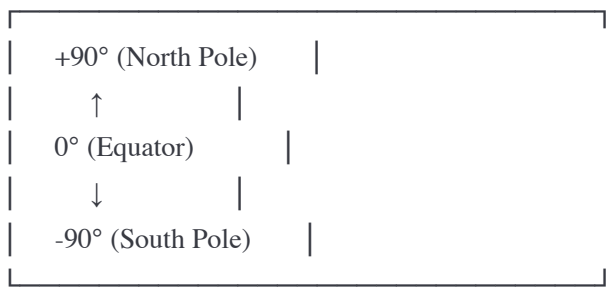
---

## 1. Geospatial Indexing

### Latitude and Longitude Basics

Coordinate System:

Latitude: North/South (-90° to +90°)




Longitude: East/West (-180° to +180°)

Example Coordinates:
New York: (40.7128° N, -74.0060° W)
London: (51.5074° N, -0.1278° W)
Tokyo: (35.6762° N, 139.6503° E)

Distance Calculation (Haversine Formula):
Takes into account Earth's curvature

---

### Haversine Distance

python

```python
import math

def haversine_distance(lat1, lon1, lat2, lon2):
    """
    Calculate distance between two points on Earth
    Returns distance in kilometers
    """
    # Earth's radius in kilometers
    R = 6371

    # Convert to radians
    lat1_rad = math.radians(lat1)
    lat2_rad = math.radians(lat2)
    delta_lat = math.radians(lat2 - lat1)
    delta_lon = math.radians(lon2 - lon1)

    # Haversine formula
    a = (math.sin(delta_lat / 2) ** 2 +
         math.cos(lat1_rad) * math.cos(lat2_rad) *
         math.sin(delta_lon / 2) ** 2)

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    distance = R * c

    return distance

# Example
ny = (40.7128, -74.0060)
london = (51.5074, -0.1278)

distance = haversine_distance(ny[0], ny[1], london[0], london[1])
print(f"New York to London: {distance:.2f} km")
# Output: 5570.25 km

# Find nearby (naive - slow for large datasets)
def find_nearby_naive(user_lat, user_lon, radius_km, places):
    """Slow: Checks every place"""
    nearby = []

    for place in places:
        distance = haversine_distance(
            user_lat, user_lon,
            place['lat'], place['lng']
        )
```

```
        if distance <= radius_km:
            nearby.append({
                'place': place,
                'distance': distance
            })

    # Sort by distance
    nearby.sort(key=lambda x: x['distance'])

    return nearby

# For 1 million places: Very slow! (calculates 1M distances)
```

---

## 2. Geohashing

### What is Geohashing?

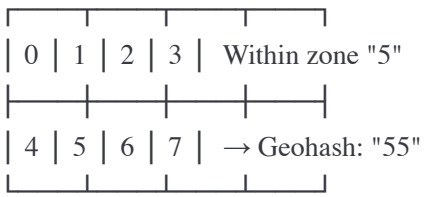
### Encode geographic coordinates into short string.

Geohash: Divide world into grid, recursively subdivide

Level 1: Divide world into 32 zones (5 bits)


(Simplified to 8 for illustration)

Level 2: Subdivide each zone into 32 sub-zones




More levels = higher precision

Geohash Levels and Precision:

| Level | Size | Example |
|---|---|---|
| 1 | ±2500 km | "9" |
| 2 | ±630 km | "9q" |
| 3 | ±78 km | "9q8" |
| 4 | ±20 km | "9q8y" |
| 5 | ±2.4 km | "9q8yy" |
| 6 | ±610 m | "9q8yyv" |
| 7 | ±76 m | "9q8yyvz" |
| 8 | ±19 m | "9q8yyvzu" |
| 9 | ±2.4 m | "9q8yyvzug" |

**Example:**

Coordinates: (40.7128, -74.0060) → Geohash: "dr5ru6r"

Nearby places with same prefix:
- "dr5ru6r" (exact location)
- "dr5ru6x" (very close)
- "dr5ru6w" (very close)
- "dr5ru7" (close)
- "dr5ru" (nearby)

Places with "dr5ru" prefix are within ~2.4 km!

Search Strategy:
1. Get geohash of user location
2. Find all places with matching prefix
3. Much faster than calculating all distances!

---

### Geohash Implementation

python

```python
class Geohash:
    # Base32 alphabet
    BASE32 = "0123456789bcdefghjkmnpqrstuvwxyz"

    @staticmethod
    def encode(latitude, longitude, precision=9):
        """
        Encode coordinates to geohash
        precision: number of characters (1-12)
        """
        lat_range = [-90.0, 90.0]
        lon_range = [-180.0, 180.0]

        geohash = []
        bits = 0
        bit = 0
        ch = 0

        while len(geohash) < precision:
            if bit % 2 == 0:  # Even bits: longitude
                mid = (lon_range[0] + lon_range[1]) / 2
                if longitude > mid:
                    ch |= (1 << (4 - bits))
                    lon_range[0] = mid
                else:
                    lon_range[1] = mid
            else:  # Odd bits: latitude
                mid = (lat_range[0] + lat_range[1]) / 2
                if latitude > mid:
                    ch |= (1 << (4 - bits))
                    lat_range[0] = mid
                else:
                    lat_range[1] = mid

            bits += 1
            bit += 1

            if bits == 5:
                geohash.append(Geohash.BASE32[ch])
                bits = 0
                ch = 0

        return ''.join(geohash)

    @staticmethod
    def decode(geohash):
```

```
    """Decode geohash to coordinates"""
    lat_range = [-90.0, 90.0]
    lon_range = [-180.0, 180.0]

    is_lon = True

    for char in geohash:
        idx = Geohash.BASE32.index(char)

        for i in range(4, -1, -1):
            bit = (idx >> i) & 1

            if is_lon:
                mid = (lon_range[0] + lon_range[1]) / 2
                if bit == 1:
                    lon_range[0] = mid
                else:
                    lon_range[1] = mid
            else:
                mid = (lat_range[0] + lat_range[1]) / 2
                if bit == 1:
                    lat_range[0] = mid
                else:
                    lat_range[1] = mid

            is_lon = not is_lon

    lat = (lat_range[0] + lat_range[1]) / 2
    lon = (lon_range[0] + lon_range[1]) / 2

    return lat, lon

@staticmethod
def neighbors(geohash):
    """Get neighboring geohashes"""
    # Simplified: Return adjacent cells
    # In production: Use complete neighbor algorithm

    base_len = len(geohash) - 1
    base = geohash[:base_len]

    last_char = geohash[-1]
    idx = Geohash.BASE32.index(last_char)

    neighbors = []

    # Get surrounding characters (simplified)
```

```
        for offset in [-1, 0, 1]:
            new_idx = (idx + offset) % 32
            neighbors.append(base + Geohash.BASE32[new_idx])

        return neighbors

# Usage
# Encode
geohash = Geohash.encode(40.7128, -74.0060, precision=7)
print(f"Geohash: {geohash}")  # dr5ru6r

# Decode
lat, lon = Geohash.decode(geohash)
print(f"Coordinates: ({lat:.4f}, {lon:.4f})")

# Find nearby
user_geohash = "dr5ru6r"
nearby_geohashes = Geohash.neighbors(user_geohash)
print(f"Nearby geohashes: {nearby_geohashes}")

# Database query (PostgreSQL with geohash index)
# CREATE INDEX idx_restaurant_geohash ON restaurants(geohash);

# Find nearby restaurants
query = """
SELECT * FROM restaurants
WHERE geohash LIKE 'dr5ru%'  -- Same prefix = nearby
ORDER BY distance(lat, lng, $1, $2)
LIMIT 20
"""

# Much faster! Uses index, not full scan
```

---

### Redis Geospatial Commands

javascript

```
const redis = require('redis');
const client = redis.createClient();

// Add locations
await client.geoadd('restaurants',
  -74.0060, 40.7128, 'restaurant-1',  // NYC
  -73.9352, 40.7306, 'restaurant-2',  // Brooklyn
  -118.2437, 34.0522, 'restaurant-3'  // LA
);

// Find within radius
const nearby = await client.georadius(
  'restaurants',
  -74.0060, 40.7128,  // User location (NYC)
  5,  // Radius
  'km',  // Units
  'WITHDIST',  // Include distance
  'ASC'  // Sort by distance
);

console.log(nearby);
// Output:
// [
//   ['restaurant-1', '0.0000'],
//   ['restaurant-2', '4.2345']
// ]
// (LA restaurant not included - too far)

// Get distance between two points
const distance = await client.geodist(
  'restaurants',
  'restaurant-1',
  'restaurant-2',
  'km'
);

console.log(`Distance: ${distance} km`);

// Get coordinates
const position = await client.geopos('restaurants', 'restaurant-1');
console.log(position);  // [[-74.0060, 40.7128]]
```

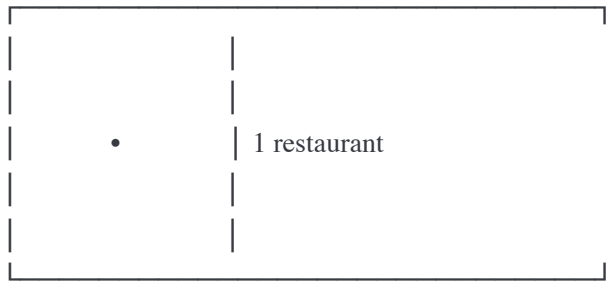
# 3. QuadTree

### What is a QuadTree?

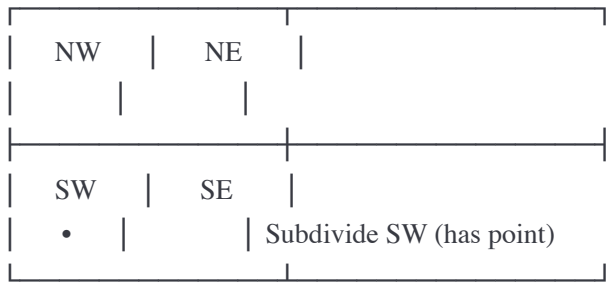
### Hierarchical spatial data structure.

QuadTree: Recursively divide space into 4 quadrants

Level 0 (World):




Level 1 (Divide into 4):




Level 2 (Subdivide SW):


Continue subdividing until each region has few points

### Properties:

Benefits:

✓ O(log n) search for nearby points

✓ Dynamic (can insert/delete)

✓ Good for changing data

Drawbacks:

✗ Uneven distribution (all points in one quadrant)

✗ Rebalancing needed

✗ Complex implementation

When to use:

✓ Dynamic data (Uber drivers moving)

✓ Need fast updates

✓ Uneven geographic distribution

---

## QuadTree Implementation

python

```
class Point:
    def __init__(self, x, y, data):
        self.x = x  # Longitude
        self.y = y  # Latitude
        self.data = data  # Restaurant/driver info

class Rectangle:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height

    def contains(self, point):
        return (self.x <= point.x < self.x + self.width and
                self.y <= point.y < self.y + self.height)

    def intersects(self, other):
        return not (other.x > self.x + self.width or
                    other.x + other.width < self.x or
                    other.y > self.y + self.height or
                    other.y + other.height < self.y)

class QuadTree:
    def __init__(self, boundary, capacity=4):
        self.boundary = boundary
        self.capacity = capacity
        self.points = []
        self.divided = False

        # Four quadrants (created when subdividing)
        self.northwest = None
        self.northeast = None
        self.southwest = None
        self.southeast = None

    def insert(self, point):
        # Check if point is in this quad
        if not self.boundary.contains(point):
            return False

        # If capacity not reached, add here
        if len(self.points) < self.capacity:
            self.points.append(point)
            return True
```

```
    # Subdivide if needed
    if not self.divided:
        self.subdivide()

    # Insert into appropriate quadrant
    return (self.northwest.insert(point) or
            self.northeast.insert(point) or
            self.southwest.insert(point) or
            self.southeast.insert(point))

def subdivide(self):
    x = self.boundary.x
    y = self.boundary.y
    w = self.boundary.width / 2
    h = self.boundary.height / 2

    # Create four quadrants
    nw = Rectangle(x, y, w, h)
    ne = Rectangle(x + w, y, w, h)
    sw = Rectangle(x, y + h, w, h)
    se = Rectangle(x + w, y + h, w, h)

    self.northwest = QuadTree(nw, self.capacity)
    self.northeast = QuadTree(ne, self.capacity)
    self.southwest = QuadTree(sw, self.capacity)
    self.southeast = QuadTree(se, self.capacity)

    self.divided = True

def query(self, range_rect, found=None):
    """Find all points within range"""
    if found is None:
        found = []

    # Check if range intersects this quad
    if not self.boundary.intersects(range_rect):
        return found

    # Check points in this quad
    for point in self.points:
        if range_rect.contains(point):
            found.append(point)

    # Recursively search subdivisions
    if self.divided:
        self.northwest.query(range_rect, found)
        self.northeast.query(range_rect, found)
```

```
            self.southwest.query(range_rect, found)
            self.southeast.query(range_rect, found)

        return found

    def query_radius(self, center_x, center_y, radius_km):
        """Find all points within radius (km)"""
        # Create bounding box
        # Approximate: 1 degree ≈ 111 km
        deg_radius = radius_km / 111

        range_rect = Rectangle(
            center_x - deg_radius,
            center_y - deg_radius,
            deg_radius * 2,
            deg_radius * 2
        )

        # Get points in bounding box
        candidates = self.query(range_rect)

        # Filter by actual distance
        nearby = []
        for point in candidates:
            distance = haversine_distance(
                center_y, center_x,
                point.y, point.x
            )

            if distance <= radius_km:
                nearby.append({
                    'point': point,
                    'distance': distance
                })

        # Sort by distance
        nearby.sort(key=lambda x: x['distance'])

        return nearby

# Usage
# Create QuadTree covering New York area
boundary = Rectangle(-74.1, 40.6, 0.3, 0.3)  # NYC area
qtree = QuadTree(boundary, capacity=4)

# Insert restaurants
restaurants = [
```

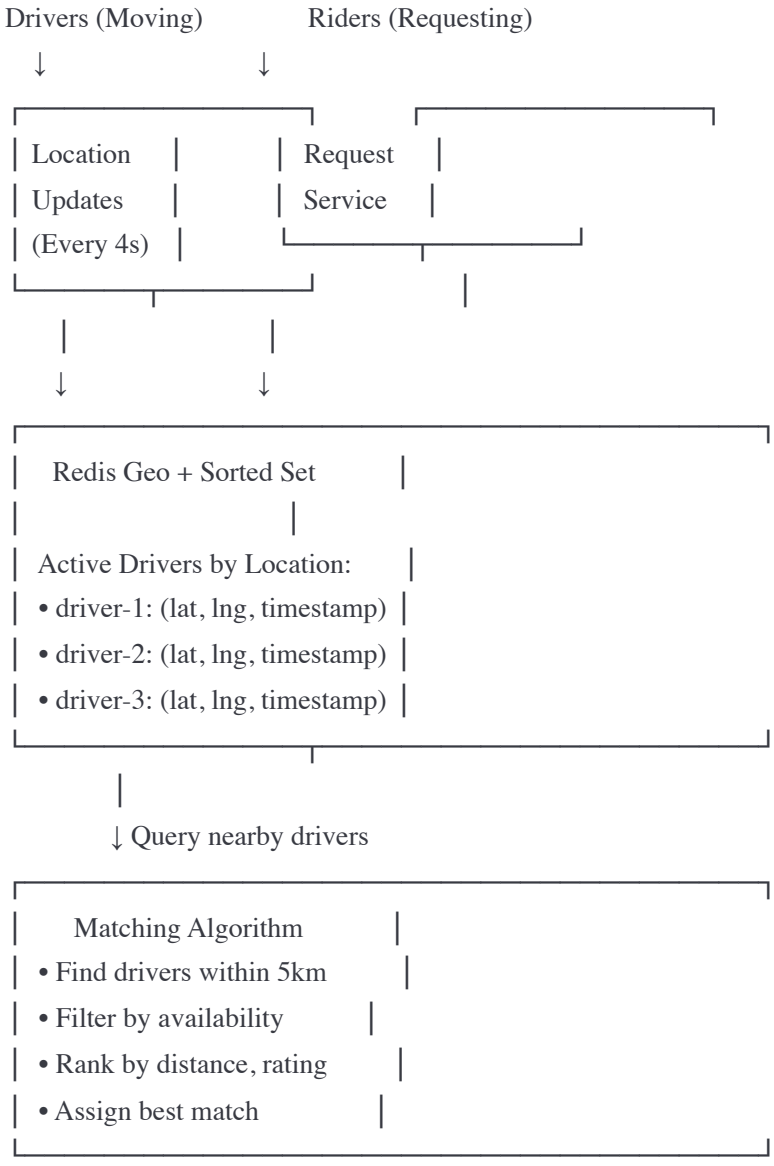
```
    Point(-74.0060, 40.7128, {'name': 'Restaurant A'}),
    Point(-73.9352, 40.7306, {'name': 'Restaurant B'}),
    Point(-74.0100, 40.7200, {'name': 'Restaurant C'}),
    # ... add thousands more
]

for restaurant in restaurants:
    qtree.insert(restaurant)

# Find nearby restaurants (5km radius)
user_location = (-74.0060, 40.7128)
nearby = qtree.query_radius(user_location[0], user_location[1], 5)

print(f"Found {len(nearby)} restaurants within 5km:")
for item in nearby[:5]:
    print(f"  {item['point'].data['name']}: {item['distance']:.2f}km")

# Output:
# Found 15 restaurants within 5km:
#   Restaurant A: 0.00km
#   Restaurant C: 0.85km
#   Restaurant B: 4.23km
#   ...
```

---

## 4. Uber-Style Location Tracking

### Real-Time Driver Location Updates:

Architecture:




---

## Implementation:

javascript

```
class UberLocationService {
  constructor(redis) {
    this.redis = redis;
    this.driverKey = 'drivers:active';
  }

  async updateDriverLocation(driverId, lat, lng) {
    const timestamp = Date.now();

    // Store in Redis Geo
    await this.redis.geoadd(
      this.driverKey,
      lng, lat, driverId
    );

    // Store additional metadata in hash
    await this.redis.hset(
      `driver:${driverId}`,
      'lat', lat,
      'lng', lng,
      'timestamp', timestamp,
      'available', 'true'
    );

    // Set expiry (if no update in 60s, remove)
    await this.redis.expire(`driver:${driverId}`, 60);

    console.log(`Updated driver ${driverId}: (${lat}, ${lng})`);
  }

  async findNearbyDrivers(userLat, userLng, radiusKm = 5, limit = 10) {
    // Find drivers within radius
    const drivers = await this.redis.georadius(
      this.driverKey,
      userLng, userLat,
      radiusKm, 'km',
      'WITHDIST',
      'ASC',
      'COUNT', limit
    );

    const results = [];

    for (const [driverId, distance] of drivers) {
      // Get driver details
      const info = await this.redis.hgetall(`driver:${driverId}`);
```

```
    if (info.available === 'true') {
      results.push({
        driverId,
        distance: parseFloat(distance),
        lat: parseFloat(info.lat),
        lng: parseFloat(info.lng),
        lastUpdate: new Date(parseInt(info.timestamp))
      });
    }
  }

  return results;
}

async assignDriver(riderId, driverId) {
  // Mark driver as unavailable
  await this.redis.hset(`driver:${driverId}`, 'available', 'false');

  // Create ride
  const rideId = this.generateRideId();

  await this.redis.hset(
    `ride:${rideId}`,
    'rider_id', riderId,
    'driver_id', driverId,
    'status', 'assigned',
    'created_at', Date.now()
  );

  return rideId;
}

async trackRide(rideId) {
  // Get ride info
  const ride = await this.redis.hgetall(`ride:${rideId}`);

  // Get driver location
  const location = await this.redis.geopos(
    this.driverKey,
    ride.driver_id
  );

  return {
    rideId,
    driverId: ride.driver_id,
    riderId: ride.rider_id,
```

```
      driverLocation: location[0],
      status: ride.status
    };
  }

  generateRideId() {
    return `ride_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
  }
}

// Usage
const locationService = new UberLocationService(redisClient);

// Drivers send location updates every 4 seconds
setInterval(() => {
  // Simulate driver movement
  const lat = 40.7128 + (Math.random() - 0.5) * 0.01;
  const lng = -74.0060 + (Math.random() - 0.5) * 0.01;

  locationService.updateDriverLocation('driver-123', lat, lng);
}, 4000);

// Rider requests ride
const riderLocation = { lat: 40.7128, lng: -74.0060 };

// Find nearby drivers
const nearbyDrivers = await locationService.findNearbyDrivers(
  riderLocation.lat,
  riderLocation.lng,
  5,  // 5km radius
  10  // Max 10 drivers
);

console.log(`Found ${nearbyDrivers.length} nearby drivers:`);
nearbyDrivers.forEach(driver => {
  console.log(`  Driver ${driver.driverId}: ${driver.distance.toFixed(2)}km away`);
});

// Assign closest available driver
if (nearbyDrivers.length > 0) {
  const closestDriver = nearbyDrivers[0];
  const rideId = await locationService.assignDriver('rider-456', closestDriver.driverId);

  console.log(`Assigned driver ${closestDriver.driverId} to ride ${rideId}`);
}
```

## 5. Real-Time ETA Calculation

### Calculate arrival time dynamically.

javascript

```
class ETAService {
  constructor() {
    this.trafficData = new Map();
    this.historicalData = new Map();
  }

  async calculateETA(fromLat, fromLng, toLat, toLng) {
    // Get route from mapping service
    const route = await this.getRoute(fromLat, fromLng, toLat, toLng);

    // Calculate base time (distance / average speed)
    const distanceKm = route.distance;
    const baseSpeedKmh = 40;  // Average city speed
    const baseTimeMinutes = (distanceKm / baseSpeedKmh) * 60;

    // Adjust for current traffic
    const trafficMultiplier = await this.getTrafficMultiplier(route);

    // Adjust for time of day
    const timeOfDayMultiplier = this.getTimeOfDayMultiplier();

    // Calculate final ETA
    const etaMinutes = baseTimeMinutes * trafficMultiplier * timeOfDayMultiplier;

    return {
      distanceKm,
      etaMinutes: Math.round(etaMinutes),
      arrivalTime: new Date(Date.now() + etaMinutes * 60000),
      confidence: this.calculateConfidence(route)
    };
  }

  async getRoute(fromLat, fromLng, toLat, toLng) {
    // Call Google Maps API or similar
    // Returns: { distance, segments, duration }

    // Simplified
    const distance = haversine_distance(fromLat, fromLng, toLat, toLng);

    return {
      distance,
      segments: this.generateSegments(fromLat, fromLng, toLat, toLng)
    };
  }

  async getTrafficMultiplier(route) {
```

```
    // Check real-time traffic on route segments
    let totalDelay = 0;

    for (const segment of route.segments) {
      const traffic = this.trafficData.get(segment.id);

      if (traffic) {
        // traffic.congestion: 0 (free flow) to 1 (standstill)
        totalDelay += segment.length * traffic.congestion;
      }
    }

    const avgCongestion = totalDelay / route.distance;

    // Multiplier: 1.0 (no traffic) to 2.5 (heavy traffic)
    return 1.0 + (avgCongestion * 1.5);
  }

  getTimeOfDayMultiplier() {
    const hour = new Date().getHours();

    // Rush hours (7-9 AM, 5-7 PM)
    if ((hour >= 7 && hour <= 9) || (hour >= 17 && hour <= 19)) {
      return 1.3;  // 30% slower
    }

    // Late night (11 PM - 5 AM)
    if (hour >= 23 || hour <= 5) {
      return 0.8;  // 20% faster
    }

    return 1.0;  // Normal
  }

  calculateConfidence(route) {
    // Based on data availability
    // More historical data = higher confidence
    return 0.85;  // 85% confidence
  }

  async updateTraffic(segmentId, congestionLevel) {
    this.trafficData.set(segmentId, {
      congestion: congestionLevel,
      timestamp: Date.now()
    });
  }
}
```

```
// Usage
const etaService = new ETAService();

// Calculate ETA
const eta = await etaService.calculateETA(
  40.7128, -74.0060,  // From: NYC
  40.7589, -73.9851   // To: Times Square
);

console.log(`Distance: ${eta.distanceKm.toFixed(2)} km`);
console.log(`ETA: ${eta.etaMinutes} minutes`);
console.log(`Arrival: ${eta.arrivalTime.toLocaleTimeString()}`);
console.log(`Confidence: ${(eta.confidence * 100).toFixed(0)}%`);

// Update ETA in real-time as driver moves
setInterval(async () => {
  const driverLocation = await getDriverLocation(driverId);
  const destination = await getRideDestination(rideId);

  const updatedETA = await etaService.calculateETA(
    driverLocation.lat, driverLocation.lng,
    destination.lat, destination.lng
  );

  // Push to rider via WebSocket
  socket.emit('eta_update', {
    etaMinutes: updatedETA.etaMinutes,
    distanceKm: updatedETA.distanceKm
  });
}, 10000);  // Update every 10 seconds
```

---

## Key Takeaways

1. **Geospatial Indexing:**
   - Geohashing: String-based, prefix matching
   - QuadTree: Hierarchical, dynamic
   - R-Tree: Advanced, production databases
   - Redis Geo: Built-in commands
2. **Real-Time Location:**
   - Frequent updates (4-10 seconds)

- Redis for fast queries
- QuadTree for moving objects
- WebSocket for push updates

3. **Proximity Search:**
   - O(log n) with spatial index
   - vs O(n) without index
   - 100-1000x faster

4. **Uber Architecture:**
   - Driver location: Redis Geo
   - Matching: QuadTree + business logic
   - ETA: Real-time traffic + historical
   - Scaling: Sharding by geography

## Practice Problems

1. Design Uber (driver matching, real-time tracking, ETA)
2. Design Yelp (nearby restaurants, search, reviews)
3. Calculate: 1M drivers updating every 4s. What throughput needed?

Ready for Chapter 23: Recommendation Systems?