HLD Popular interview questions

Contents

[Designing a system like Uber 2](#_Toc184851738)

[Quad Trees and Applications 3](#_Toc184851739)

[Key Architectural Concepts 6](#_Toc184851740)

[Database Concepts and Use Cases 9](#_Toc184851741)

[Uber Regional systems 11](#_Toc184851742)

[Bootstrapping the Quadtree at a new location 13](#_Toc184851743)

[Uber System Design Using Quad Trees 16](#_Toc184851744)

[Analysis of Driver Location Flow: 18](#_Toc184851745)

[Analysis of Ride Booking Flow 20](#_Toc184851746)

[Notifying Drivers About Rides 22](#_Toc184851747)

[Alternative Notification Mechanisms 25](#_Toc184851748)

[Notification and Driver Location Management 28](#_Toc184851749)

## Designing a system like Uber

**Designing a system like Uber involves systematically addressing user needs, scale, and architectural trade-offs. Here's a structured way to approach it:**

**1. Identify Core Features**

**User Features**

* Book a cab or request a ride.
* View ride history and estimated time of arrival.
* Real-time driver tracking.
* Fare estimation and payment options.

**Driver Features**

* Accept or decline ride requests.
* View ride history and earnings.
* Navigation assistance for pickup and drop-off locations.

**Additional Features**

* Ride matching algorithm to optimize driver-user pairings.
* Notifications for users and drivers (e.g., ride updates).
* Multi-city or global availability.
* Customer support and feedback mechanisms.

**2. Estimate Scale**

**Drivers**

* For a big city like Delhi, assume ~50,000–100,000 drivers.
* Global scale: ~10–20 million drivers across ~100 large cities.

**Users**

* Assume 2% of the global population (~140 million) as potential active users, aligning with Uber's 2023 numbers.

**3. Address Scale Challenges**

**Driver-User Matching**

* Efficient ride-matching algorithms with low latency.
* Ensure drivers nearby are matched to minimize wait times.

**Data Management**

* Real-time updates for driver and user locations.
* Large-scale storage for ride history and user preferences.
* Partitioned databases for regional scalability.

**Global Infrastructure**

* Multi-region deployments to reduce latency.
* Load balancers to handle peak traffic efficiently.

**4. Design Trade-Offs**

**Latency vs. Availability**

* **Matching Algorithm:** Prioritize low latency to ensure a seamless experience.
* **Data Consistency:** Sacrifice strict consistency for availability in real-time features (e.g., driver location updates).
* **High Availability:** Focus on uptime for critical services like booking and payments.

**Consistency Requirements**

* Critical financial transactions (e.g., fare calculations and payments) require strong consistency.
* Ride status updates can tolerate eventual consistency.

## Quad Trees and Applications

**Problem Context:**

1. **Objective:**
   * Create a system to answer location-based queries, such as finding nearby points of interest (e.g., temples, restaurants, cabs).
   * Examples include Google Maps, Yelp, or Uber.
2. **Challenges:**
   * Efficiently handle queries about nearby locations.
   * Update the system dynamically as locations are added or removed.

**Solution Approach: Quad Trees**

**What is a Quad Tree?**

* A tree-like data structure used to divide a 2D space into variable-sized grid cells.
* Each cell (node) contains:
  + **Boundary coordinates**: Defined by four XY points.
  + **Cell ID**: Unique identifier.
  + **List of locations**: Popular places or items in that cell.

**Steps to Build a Quad Tree (Precompute Phase):**

1. **Start with the Entire Space:**
   * Assume the whole world as one grid cell (node).
2. **Set a Threshold (T):**
   * E.g., each cell can hold up to 100 places of interest.
3. **Recursive Division:**
   * For a cell with more than T places:
     + Divide it into four smaller, equal-sized cells.
   * Repeat the process for each sub-cell until all cells meet the threshold.

**Key Properties of Quad Trees:**

* **Variable-sized grid cells:**
  + Densely populated regions (e.g., cities) have smaller cells (higher tree depth).
  + Sparsely populated regions (e.g., deserts) have larger cells (lower tree depth).
* **Leaf Nodes:**
  + Represent the smallest cells that meet the threshold.
  + Contain the actual places of interest.

**Handling Queries:**

1. **Find Nearby Places:**
   * Locate the user in the corresponding grid cell.
   * Search that cell and neighbouring cells for places of interest.
2. **Dynamic Updates:**
   * Add or remove locations:
     + Adjust the relevant cell's data.
     + **Additions**:
       - If the count exceeds T, split the cell into smaller cells.
     + **Removals**:
       - Check if the cell and its siblings can merge back into a larger cell.

**Application to Uber-like Systems:**

**Similarity to Location Systems:**

* Treat cabs (drivers) as "places of interest."
* Quad trees can efficiently handle queries to find nearby cabs.

**Key Differences:**

1. **Dynamic Nature:**
   * Unlike fixed places (e.g., landmarks), cabs move frequently.
   * This requires constant updates:
     + Remove the cab from the old cell.
     + Add it to the new cell.
2. **Frequent Cell Adjustments:**
   * Cells may:
     + Quickly fill up (e.g., during office hours near business complexes).
     + Become sparse (e.g., during evenings when cabs spread out).
   * Dynamic division and merging of cells are necessary to handle these variations.

**Examples of Dynamic Updates:**

1. **Office Hours:**
   * Cabs converge toward business hubs, making cells denser.
   * Requires frequent splitting of these cells.
2. **Evenings:**
   * Cabs disperse from hubs, reducing density.
   * Allows merging of cells.

**Conclusion:**

* Quad trees provide an efficient way to manage both static and dynamic location-based data.
* By adapting quad tree techniques to handle moving objects, they can support applications like Uber.
* Key considerations:
  + Optimize for frequent updates due to object movement.
  + Balance cell splitting and merging to maintain performance.

## Key Architectural Concepts

1. **Client Interaction**:
   * Clients can be browsers, mobile apps (iOS/Android), or desktop apps.
   * Requests are routed via DNS to the system's entry point, typically a **gateway**.
2. **Gateway**:
   * Acts as the first point of contact for external clients.
   * Often doubles as a load balancer and provides a single-entry point to protect internal systems.
3. **Backend Components**:
   * **App Servers**: Handle business logic. These can be **stateful** or **stateless**.
   * **Databases**:
     + **User Database (UDB)**: Stores user and driver profiles (structured data).
     + Other specialized databases may include transactional databases, graph databases, etc., forming a **database federation**.
   * **Message Queues**: Facilitate asynchronous communication between services.
   * **Caches**: Used for performance optimization (e.g., global caches, app-level caches).
   * A computer network diagram with different devices

     Description automatically generated with medium confidence**CDNs**: Optimize content delivery for static assets.
4. **Cold Storage**:
   * Stores infrequently accessed data (e.g., archive data).
   * Cost-effective and used for regulatory or forensic purposes.
5. **Quadtree for Location Data**:
   * Efficient data structure for storing geographical points of interest.
   * Fits within a single machine for high-performance queries.
6. **Localized Subsystems**:
   * Uber operates in geographic silos; servers in a specific region handle requests for that region.
   * This minimizes latency and prevents unnecessary burden on the global system.

**Next Steps in Designing Uber**

**System Design Details:**

1. **Modular Services**:
   * Identify services required (e.g., User Service, Ride Matching Service, Payment Service).
   * Ensure they communicate effectively through APIs.
2. **Database Federation**:
   * Select appropriate databases for different services:
     + **SQL for structured data** (UDB, Payments).
     + **Quadtree for location data**.
     + **NoSQL (e.g., MongoDB)** for semi-structured data like ride history or chat messages.
3. **Ride Matching Logic**:
   * Implement algorithms to match users with nearby drivers.
   * Use the quadtree to find drivers efficiently.
4. **Real-Time Systems**:
   * Employ **WebSockets** or **long polling** for real-time communication between clients and backend for live ride tracking.
5. **Scalability**:
   * Use a **microservices architecture** to scale individual components independently.
   * Add **horizontal scaling** for app servers, caches, and databases.
6. **Fault Tolerance**:
   * Employ failover mechanisms for critical components.
   * Use **replication** for databases and **distributed systems** for message queues (e.g., Kafka).

## Database Concepts and Use Cases

**1. Sharding**

* **Definition**:
  + The process of distributing data across multiple machines when it cannot fit into a single machine.
  + Each machine holds a part of the data.
  + Data points are mutually exclusive, i.e., no data point exists in more than one shard.
* **Example**:
  + User DB containing 200 million users and drivers.
  + This database would be sharded into multiple machines for storage and management.
* **Key Points**:
  + Sharding is implemented to distribute and manage large datasets effectively.
  + It ensures scalability and efficiency.

**2. Federation of Databases**

* **Definition**:
  + Instead of sharding a single database, multiple independent databases are created and used for specific regions or use cases.
* **Difference from Sharding**:
  + Sharding involves dividing a single dataset across machines.
  + Federation involves independent databases for different regions or purposes.
* **Example**:
  + Quadtree Database:
    - Stores nearby places of interest (e.g., cab matching regions).
    - Independent quadtrees for regions like Delhi, Mumbai, Jaipur, etc.
    - Not a single quadtree sharded across machines but multiple independent quadtrees.
* **Handling Border Cases**:
  + Regions are divided based on factors like connectivity and population density, not strict city boundaries.
  + Drivers moving between regions are updated in the appropriate quadtree.

**3. Example Databases for Uber-Like Applications**

**a. User Database (User DB)**

* Purpose: Stores user and driver information.
* Implementation: Typically, sharded for scalability.

**b. Quadtree Database**

* Purpose: Solves "nearby places of interest" problems, such as cab matching.
* Characteristics:
  + Independent quadtrees per region.
  + No sharding of a single quadtree.

**c. Trips Database**

* Purpose: Stores structured information about trips.
* Example Schema:
  + Trip ID, User ID, Pickup Location, Drop Location, Fare, etc.
* Recommended DB Type: SQL.
* Sharding: Possible based on location (e.g., Delhi vs. the United States).

**d. Detailed Location Tracker**

* Purpose: Tracks drivers’ locations over time.
* Data Use:
  + Analytics for driver activity and suggestions for better business opportunities.
  + Retention of historical data for future analysis.
* Recommended DB Type: Wide Column Family Databases (e.g., Cassandra).
  + Advantages:
    - Efficient for time-series data.
    - Querying historical driver locations (e.g., last 10 positions).

**4. Importance of Data**

* **Data as the New Oil**:
  + Companies like Uber must retain rich data instead of deleting it.
  + Data drives analytics and business insights.
* **Examples of Data Utilization**:
  + Understanding driver activity.
  + Suggesting optimal regions for business.
  + Future-proofing business analytics by retaining historical data.

**5. Key Concepts Covered**

* **Sharding**: Efficient distribution of large datasets.
* **Federation**: Independent databases tailored for specific use cases or regions.
* **Database Types**:
  + SQL for structured data (e.g., Trips DB).
  + Wide Column Family for time-series data (e.g., Location Tracker).
* **Data Retention**:
  + Critical for analytics and long-term business insights.

## Uber Regional systems

This discussion provides a detailed explanation of how Uber handles its regional systems and related backend architecture. Let’s summarize the key points and flow of this system:

**Key Points:**

1. **Federation of Databases**:
   * A single database may not solve all use cases; instead, a federated approach is better.
   * Each Uber region (e.g., Delhi, Mumbai, Paris) operates as an independent subsystem with its own data and servers.
2. **Regional Systems**:
   * Each region has its own **gateway machines** with unique IP addresses.
   * Clients are mapped to the nearest regional gateway through DNS.
3. **DNS Resolution**:
   * **Geo-based DNS**: Returns the IP of the closest regional gateway based on the client’s location.
   * **Waterfall DNS**: Returns a list of gateway IPs in random order, and the client attempts to connect sequentially.
   * A screenshot of a computer

     Description automatically generatedIf a client connects to the wrong gateway (e.g., Delhi client connects to Paris gateway), the gateway redirects the client to the correct one.
4. **Client Behaviour**:
   * Clients send their live location alongside requests, allowing servers to route them to the correct regional subsystem.
   * A temporary delay might occur when moving regions (e.g., flying from Delhi to Bangalore) due to client-side cache containing outdated location information.
5. **Internal Architecture of a Regional System**:
   * **Components**:
     + **Gateway**: Handles incoming requests.
     + **Load Balancer**: Distributes requests across multiple app servers.
     + **App Servers**: Process client requests and interact with databases.
     + **Databases**: Store region-specific data such as user profiles, trips, live locations.
     + **Quadtree Machine**: Dedicated for efficiently querying spatial data.
6. **Quadtree Optimization**:
   * A quadtree helps in managing spatial data like driver locations efficiently.
   * Regional quadtrees are smaller in size (e.g., a few MBs for a city) and can fit into the RAM of app servers for faster access.
   * Backup:
     + RAM holds the active quadtree.
     + A serialized version is stored on the hard disk for persistence.
     + Additional redundancy is maintained with slave machines.
7. **Serialization**:
   * Converts in-memory data structures (like quadtrees) into a format that can be saved on a disk and reconstructed later.
   * Example: Representing a heap (a graph in RAM) as a list of elements stored in the database.

## Bootstrapping the Quadtree at a new location

**1. Bootstrapping the Quadtree**

* **Objective**: Create a spatial data structure (Quadtree) for efficient region-based processing.
* **Steps**:
  1. Use reference data (e.g., population density, public transport routes, market areas, Google Maps data) to approximate initial grid sizing.
  2. Smaller grid cells for densely populated areas, larger cells for sparsely populated regions to optimize cab placement and minimize lookup times.

**2. Driver Location Updates**

* **Driver App**:
  + Periodically sends latitude and longitude (every minute) to the server using HTTP requests or WebSocket communication.
  + Includes driver ID, current status (e.g., on ride, waiting), and location details.

**3. Global Cache System**

* **Purpose**: Minimize dependency on Quadtree for frequent operations.
* **Technology**: Use a **Redis Cluster** for caching.
* **Cache Structure**:
  1. **Cache 1**:
     + **Key**: Driver ID.
     + **Value**:
       - Last recorded location (X, Y).
       - Last grid cell ID.
  2. **Cache 2**:
     + **Key**: Grid Cell ID.
     + **Value**:
       - Boundaries of the cell (coordinates: A, B, C, D).

**4. Data Flow for Location Updates**

* **Step 1**: Store new driver location in a **Location DB** for historical tracking.
* **Step 2**: Query **Cache 1** to retrieve:
  + Last recorded driver location.
  + Last grid cell ID.
* **Step 3**: Compare:
  + **Case A**: If the driver hasn’t moved, skip further processing.
  + **Case B**: If the driver moved but remains within the same grid cell, no Quadtree update is needed.
  + **Case C**: If the driver moved to a new grid cell:
    - Update Quadtree:
      1. Delete the driver from the old grid cell.
      2. Add the driver to the new grid cell.

**5. Advantages of Using the Cache**

* **Minimizes Queries**: Frequent operations (e.g., location checks) are resolved via cache without querying the Quadtree.
* **Improves Performance**: Updates to Quadtree are limited to significant events (e.g., grid cell changes).
* **Reduces Latency**: Cache lookups are faster than Quadtree queries, ensuring quicker responses.

**6. Quadtree Updates**

* **When to Update**:
  + Driver moves out of the current grid cell.
* **Operations**:
  + Delete driver from the old cell.
  + Insert driver into the new cell.

**7. Design Considerations**

* **Stateless App Servers**:
  + Handle driver location updates in a round-robin or load-balanced manner.
* **Dual Mechanism** (HTTP & WebSocket):
  + HTTP for periodic updates.
  + WebSocket for real-time, low-latency communication.

**8. Optimization Goals**

* Reduce traffic to Quadtree.
* Store only essential data in cache for quick decision-making.
* Balance accuracy with performance for a scalable system.

## Uber System Design Using Quad Trees

**Overview of the System**

1. **Regional Subsystems**:
   * The system is divided into subsystems, with each region having its own independent subsystem.
   * Each subsystem manages an independent quad tree for its region.
2. **Caching with Redis**:
   * A Redis global cache cluster is utilized alongside slave/read replicas of the quad tree machine.
   * Purpose:
     + Minimize write requests to the quad tree.
     + Avoid overloading the master machine responsible for writes.
   * Writes occur only on the master machine of the quad tree.
3. **Challenges with Quad Tree Dynamics**:
   * Continuous merging and splitting of quad tree cells due to fluctuating driver densities.
   * Cells:
     + Merge when driver density decreases.
     + Split when driver density increases.

**Solutions to Frequent Merging and Splitting**

**1. Range-Based Threshold**

* **Idea**:
  + Use upper and lower thresholds instead of a single-point threshold.
* **How it Works**:
  + **Split**:
    - Occurs only when the number of drivers in a cell exceeds the upper threshold.
  + **Merge**:
    - Occurs only when the number of drivers in all child cells drops below the lower threshold.
* **Benefits**:
  + Prevents frequent toggling between splitting and merging due to small fluctuations in driver numbers.
  + Adds a buffer, ensuring changes only happen when there are significant variations.

**2. Periodicity-Based Updates**

* **Idea**:
  + Adjust grid cells at regular intervals rather than continuously.
* **How it Works**:
  + The system checks the density data and adjusts cell boundaries after a set time interval (e.g., every hour or every two hours).
  + Assumes that significant changes in density take time to manifest (e.g., office hours, events).
* **Benefits**:
  + Reduces the frequency of cell boundary updates.
  + Ensures fewer system updates while maintaining reasonable accuracy.
  + Accepts a minor inefficiency in cell division during the interval for overall system stability.

**Key Observations**

1. **Dynamic Nature of Driver Density**:
   * The density of drivers changes throughout the day, influenced by patterns like office hours or events.
2. **Trade-offs in System Design**:
   * Slightly inaccurate cell divisions during intervals are acceptable for reducing the frequency of system updates.
   * The use of range thresholds minimizes unnecessary operations caused by small fluctuations.
3. **Optimization Goals**:
   * Minimize system overhead due to continuous cell merging and splitting.
   * Maintain efficiency and scalability in handling dynamic driver densities.

## Analysis of Driver Location Flow:

**Components Overview:**

1. **Clients and Load Balancer:**
   * Clients (mobile/browser) connect to a gateway/load balancer.
   * The load balancer directs traffic to appropriate application servers.

A diagram of a computer network

Description automatically generated

1. **Application Servers:**
   * Stateless in design, enabling scalability and reliability.
   * Handle client requests and delegate to back-end services.
2. **Redis Cache:**
   * Acts as a high-speed data access layer.
   * Stores frequently accessed or transient data to reduce database load.
3. **Quad Tree:**
   * A master-slave architecture for scalable reads.
   * Efficient for spatial or geographical data operations.
4. **Databases:**
   * **User Database (UDB):** Handles user-related data with master-slave replication for scalability.
   * **Trips Database:** Stores trip-related information, also replicated.
   * **Location Database:** Dedicated to real-time location updates.

**Driver Location Flow:**

1. **Driver Updates Location:**
   * Driver continuously sends their location to the system.
   * The app server processes the update.
2. **Location Update Path:**
   * App server updates the **Location Database** with the driver's current position.
   * Cache (Redis) may be queried or updated depending on the flow.
   * **Quad Tree** is updated for efficient spatial querying and analytics.
3. **Key Characteristics:**
   * High availability and scalability across all components.
   * Real-time location processing ensures accurate and timely data handling.
   * Efficient spatial data handling via Quad Tree architecture.

## Analysis of Ride Booking Flow

**Components and Flow Overview**

1. **User Request:**
   * User initiates a ride booking request.
   * The request is directed to the **gateway/load balancer** and routed to a random **stateless app server**.
2. **Quad Tree Lookup:**
   * App server identifies the user’s location and queries the **Quad Tree** database to find the relevant grid cell.
   * Neighbouring cells are also queried to fetch a list of potential drivers in proximity.
   * Drivers are filtered based on criteria such as:
     + Current availability (trip ended or ending soon).
     + Last reported location update (e.g., within the last 10 minutes).
     + Estimated trip completion time.
3. **Driver Filtering & Sorting:**
   * The app server filters drivers who do not qualify (e.g., poor ratings, inactivity, long trip completion times).
   * Remaining drivers are **sorted** based on parameters like proximity to the user or driver rating.
4. **Driver Polling for Matching:**
   * The app server applies a **waterfall model** for polling drivers:
     + **Sequential Polling (One by One):**
       - The top driver in the list is sent a request to accept the ride.
       - If they do not respond within a set time (e.g., 30 seconds), the next driver is polled.
       - Process continues until a driver accepts the ride.
     + **Parallel Polling (Controlled):**
       - Instead of polling one driver at a time or all at once, a batch of top drivers (e.g., 3) is polled simultaneously.
       - If no response is received, the next batch is polled.
     + **All-at-Once Polling:** (Avoided for practical reasons)
       - Overwhelms the system with simultaneous requests.
       - Multiple drivers might accept simultaneously, leading to consistency and latency issues.
5. **Notification to Drivers:**
   * Once a driver is selected, a **notification** is sent to prompt acceptance of the ride request.

**Key Considerations:**

* **Scalability:**
  + Avoid overwhelming the system with too many simultaneous requests.
  + Balance load by targeting specific drivers sequentially or in small batches.
* **Consistency and Latency:**
  + Prevent scenarios where multiple drivers accept at once, leading to delays in resolving who gets the ride.
  + Sequential or controlled parallel polling reduces these risks.
* **Driver Availability:**
  + Include logic to filter out drivers based on criteria like inactivity or unavailability for a certain period.
* **Dynamic Polling Mechanisms:**
  + Choose between fully sequential, fully parallel, or hybrid polling based on system requirements and load.

**Summary of Ride Booking Flow**

1. User sends a ride request via the gateway to a random app server.
2. App server queries the Quad Tree to find nearby active drivers.
3. Drivers are filtered and sorted based on proximity, ratings, and availability.
4. The app server polls drivers sequentially or in small batches for ride acceptance.
5. Notifications are sent to selected drivers, prompting their response.
6. Once a driver accepts, the match is finalized, and the ride is confirmed.

## Notifying Drivers About Rides

**Key Question:**

How to efficiently notify drivers about ride opportunities?

**Possible Solutions:**

1. **WebSockets:**
   * **Mechanism:**
     + A persistent, bidirectional connection is maintained between the driver and the server.
     + The server can send a message to the driver anytime, and vice versa.
   * **Advantages:**
     + Real-time communication with minimal latency.
     + Immediate notification when a ride is available.
   * **Disadvantages:**
     + High cost of maintaining persistent WebSocket connections for all active drivers.
     + Increased resource usage on app servers.
   * **Use Case:**
     + Suitable for systems where real-time communication is critical and resources are abundant.
2. **HTTP Long Polling (Preferred Approach):**
   * **Mechanism:**
     + Drivers periodically send an API request to the server asking, *"Do you have a ride for me?"*
     + Instead of responding immediately, the server waits (holds the request) until:
       1. A ride becomes available within a specified time frame (e.g., 30 seconds).
       2. The request times out (no ride available).
     + If a ride becomes available during the wait, the server replies with the ride details.
   * **Advantages:**
     + Drivers receive ride notifications in near real-time.
     + Avoids maintaining persistent WebSocket connections.
     + Efficient in reducing unnecessary API calls compared to frequent polling.
   * **Disadvantages:**
     + Requires a server thread to be held for each long polling request.
     + Slightly higher server load compared to traditional HTTP requests.

**HTTP Long Polling Workflow:**

1. **Driver Sends a Request:**
   * Driver calls the server: *"Do you have a ride for me?"*
2. **Server Behavior:**
   * **If a Ride Exists:**
     + Server immediately responds with ride details, allowing the driver to accept or reject it.
   * **If No Ride Exists:**
     + Server holds the request for up to 30 seconds (or the configured timeout).
     + If a ride becomes available within this window, the server responds with the ride details.
   * **If Timeout Occurs:**
     + Server responds with "No ride available."
     + Driver repeats the request after the timeout.
3. **Driver Perspective:**
   * Receives ride details only when a match is found.
   * Sends a new request only if the previous one times out.

**Why HTTP Long Polling is Preferred Over WebSockets:**

1. **Cost Efficiency:**
   * WebSockets require maintaining persistent connections for all active drivers, consuming more resources.
   * Long polling avoids the overhead of managing these persistent connections.
2. **Scalability:**
   * Long polling scales better in systems with a large number of active drivers by avoiding constant real-time connections.
3. **Real-Time-Like Behavior:**
   * Though not fully real-time, long polling provides near real-time responses, which are sufficient for ride-matching scenarios.
4. **Simplified Maintenance:**
   * Long polling simplifies server design by leveraging stateless HTTP requests instead of maintaining stateful WebSocket connections.

**Key Considerations in HTTP Long Polling Implementation:**

1. **Thread Usage:**
   * Each long polling request consumes a thread, which could increase server load.
   * Efficient thread management is crucial to avoid bottlenecks during peak usage.
2. **Timeout Configuration:**
   * A longer timeout window (e.g., 30 seconds) balances real-time responsiveness and system load.
3. **API Design:**
   * Drivers send location updates periodically (e.g., every 1 minute).
   * Long polling requests are separate from location updates.
4. **Fallback for Timeout:**
   * Drivers need to re-initiate the request if no ride is matched within the timeout period.

**Summary of Notification Mechanism:**

* **Drivers continuously send location updates to the server via an API.**
* To receive ride notifications:
  + Drivers use **HTTP long polling** to query for rides.
  + The server holds the request until a ride is found or a timeout occurs.
* Compared to WebSockets:
  + Long polling is more cost-effective and scalable.
  + Provides near real-time notifications without persistent connections.

## Alternative Notification Mechanisms

**1. Long Polling (Primary Approach)**

* Already discussed as the preferred approach for notifying drivers.
* Key characteristics:
  + Efficient and cost-effective.
  + Server waits for up to a set timeout (e.g., 30 seconds) before responding.
  + Near real-time notifications without maintaining a persistent connection.

**2. WebSockets (Complete Push-Based Mechanism)**

* **Mechanism:**
  + Persistent bidirectional connection is established between the driver’s app and the server.
  + Server can push notifications to the driver whenever needed.
* **Advantages:**
  + Real-time push notifications with minimal latency.
  + Suitable for highly dynamic systems where rapid communication is critical.
* **Disadvantages:**
  + Expensive due to the need to maintain persistent connections for all active drivers.
  + Resource-intensive on the server side.
* **Use Case:**
  + Ideal for systems that prioritize ultra-low latency and have the infrastructure to support it.

**3. Third-Party Notification Services (e.g., Apple Push Notification Service, Firebase Cloud Messaging)**

* **Mechanism:**
  + The app leverages the platform’s native notification services (Apple for iOS, Firebase for Android).
  + Example:
    - The app sends a request to Apple Push Notification Service (APNS) or Firebase Cloud Messaging (FCM).
    - The notification is then relayed to the driver’s device by the platform.
* **Advantages:**
  + Notifications can reach the user even when the app is not running in the background.
  + No need for the app to maintain its own infrastructure for push notifications.
* **Disadvantages:**
  + **Cost:** App owners must pay for these services.
  + **Higher Latency:** Relaying through a third-party service introduces additional delay.
  + **Dependency on External Services:** Relies on platform providers (Apple, Google/Android), which may impact reliability and availability.
* **Use Case:**
  + Best for non-critical notifications or for apps that cannot justify maintaining their own notification infrastructure.

**Comparison of Approaches**

| **Mechanism** | **Advantages** | **Disadvantages** | **Use Case** |
| --- | --- | --- | --- |
| **Long Polling** | Cost-effective, near real-time, simple to implement | Slightly higher server load per request | Preferred for most ride-matching scenarios |
| **WebSockets** | Ultra-low latency, real-time, bidirectional | High cost, resource-intensive | Critical systems needing ultra-low latency notifications |
| **Third-Party Notification Services** | Works even if app is not active, no server setup required | Costly, higher latency, third-party dependency | Non-critical or supplementary notifications |

**Key Insights from the Lecture**

1. **Flexibility in Mechanism Choice:**
   * The notification mechanism should be selected based on:
     + Cost considerations.
     + Real-time requirements.
     + Infrastructure availability.
2. **Why Long Polling is Preferred:**
   * Balances cost and performance for ride-matching notifications.
   * Avoids dependency on third-party notification systems.
3. **Role of Third-Party Notification Services:**
   * These are supplementary and not optimal for real-time notifications.
   * Suitable for cases where the app is in the background or inactive.
4. **WebSockets as a High-End Solution:**
   * WebSockets are more expensive and resource-intensive but provide unparalleled real-time responsiveness.

**Summary**

* **Primary Mechanism for Notifications:** Long polling is efficient, cost-effective, and sufficiently responsive for ride-matching notifications.
* **Fallback or Supplementary Mechanism:** Third-party notification services (like APNS or FCM) can be used when the app is not active, but they incur additional costs and latency.
* **Advanced Mechanism:** WebSockets are an option for highly responsive systems but are costlier and require robust infrastructure.

## Notification and Driver Location Management

**Notification Overview**

* Notifications play a crucial role in real-time ride-matching systems like Uber.
* Various mechanisms for notifying users and drivers include:
  1. **Long Polling** (Primary mechanism)
  2. **WebSockets**
  3. **Third-Party Notification Services** (e.g., APNS, FCM)

**Driver Location Flow**

**Driver's Responsibility**

* Drivers must continuously update their location to the server at regular intervals (e.g., every minute).
* Each update includes:
  + Driver ID
  + Current location (latitude and longitude)

**System Flow**

1. **Location Update Handling:**
   * Driver sends a location update to the gateway.
   * Gateway forwards it to a stateless app server (using round-robin or similar load balancing).
2. **Location Database (Location DB):**
   * App server updates the **Location DB**, which tracks:
     + Driver's current location.
     + Activity logs for analytics (e.g., location history throughout the day).
   * Data stored in a **wide column family store** to facilitate:
     + Analytics for driver movement patterns.
     + Insights to optimize driver operations.
3. **Cache and Quadtree Interaction:**
   * Location updates also involve **Redis Cache** and the **Quadtree** (for spatial indexing):
     + **Redis Cache** stores the last recorded location of the driver.
     + **Quadtree** is used to determine the spatial cell corresponding to the driver's location.

**Minimizing Quadtree Writes**

* **Goal:** Reduce the number of write operations to the quadtree master to optimize performance.
* **Process:**
  1. Compare the new location with the last location stored in Redis Cache.
  2. Determine if the driver has moved to a new spatial cell.
     + **Same location:** Skip updating the quadtree.
     + **Same cell ID:** Skip updating the quadtree.
     + **New cell ID:** Update the quadtree:
       - Remove driver ID from the old cell.
       - Add driver ID to the new cell.
       - Update Redis Cache with the new location.
  3. Write updates to the quadtree occur only when necessary (based on the checks above).

**Quadtree Storage**

* Stored in **RAM** for faster access (volatile storage).
* Uses **master-slave architecture**:
  + Slave replicas for read-heavy operations.
  + Master for write operations.
* Periodically serialized and saved to disk to ensure data recovery in case of system failure.

**Ride Booking Flow**

**Ride Booking Request**

1. **User Action:**
   * User submits a request to book a ride.
   * Includes:
     + Current location (pickup point).
     + Destination location.
2. **Request Flow:**
   * Sent to the **gateway**.
   * Gateway forwards it to a **random app server**.

**Driver Search**

1. **Quadtree Query:**
   * App server queries the **quadtree** for drivers:
     + Current cell ID.
     + Neighbouring cell IDs.
   * Retrieves a list of nearby drivers.
2. **Filtering Drivers:**
   * Filter out drivers who:
     + Are on long, ongoing rides.
     + Have other constraints (e.g., off-duty).
3. **Sorting Drivers:**
   * Sort remaining drivers based on:
     + Proximity to the user.
     + Driver ratings:
       - Match high-rated users with high-rated drivers.
       - Match low-rated users with relatively low-rated drivers.

**Ride Matching**

1. Notifications are sent to the shortlisted drivers to accept the ride.
   * Can be sent one-by-one or in parallel to a small group of drivers.
   * Once a driver accepts, the ride is confirmed.
2. **Driver Notification Mechanisms:**
   * **Long Polling:** Driver’s app periodically sends requests to check for new rides.
   * **WebSockets:** Persistent connection for real-time ride notifications.
   * **Third-Party Services:** Leverage Android or Apple notification services to send ride requests.

**Engineering Insights**

**Data Usage and Analytics**

* Tracking driver locations in the Location DB enables:
  + Real-time matching.
  + Post-analysis for business insights:
    - Encourage drivers to optimize their movements to maximize earnings.
    - Identify trends to improve app operations.

**Efficiency Techniques**

* Use of **Redis Cache** and **quadtree replicas** minimizes unnecessary master updates.
* Quadtree serialization ensures system reliability in case of failure.

**Notification Challenges**

* Each notification mechanism has trade-offs:
  + **Long Polling:** Cost-effective but relies on periodic driver requests.
  + **WebSockets:** Real-time but resource-intensive.
  + **Third-Party Services:** Convenient but costly and introduces latency.

**Key Takeaways**

* Efficient handling of driver location updates is critical for both real-time matching and analytics.
* Leveraging a combination of caching, quadtree replicas, and strategic writes reduces system load.
* Notification systems must balance cost, latency, and infrastructure requirements based on the application’s priorities.