Detailed System Design for Order Clustering in Quick Commerce

System Overview

This design focuses on clustering orders based on delivery locations and time constraints to optimize multi-order assignments for delivery personnel. The system uses AWS services for cloud deployment, ensuring scalability and reliability. Orders are ingested in real-time, stored in DynamoDB, and clustered periodically using DBSCAN, with results available for the order allocation system.

Data and Clustering

The system processes order metadata including dark store ID, pickup and delivery locations, and delivery deadlines. Clustering occurs every 5 minutes, filtering orders with deadlines within 30 minutes and grouping them by dark store, using DBSCAN for spatial clustering. Noise points form single-order clusters for comprehensive coverage.

Cloud Deployment

Deployed on AWS, the system uses API Gateway for ingestion, DynamoDB for storage, Lambda for clustering, and CloudWatch for scheduling. This setup ensures automatic scaling, cost efficiency, and high availability, suitable for handling variable loads during peak events.

Survey Note: System Design for Order Clustering

Introduction

This document presents a comprehensive system design for an ML-based clustering system tailored for a quick commerce company aiming to optimize order assignments through multi-order clustering. The design addresses the need to cluster orders based on geospatial and temporal metadata, ensuring efficiency in delivery time, fairness in workload distribution, and operational feasibility. The system is deployed in a cloud environment, specifically AWS, to leverage scalability and cost-effectiveness.

System Requirements and Assumptions

The system must cluster orders in real-time or near-real-time, handling varying loads such as peak hours, festivals, and sports events like cricket matches. It focuses on functional requirements like clustering based on delivery locations and temporal constraints, and non-functional requirements like scalability, high availability, and cost feasibility.

Assumptions include:

- Each dark store serves a specific geographical area, with orders linked to these stores.
- Orders include essential metadata: pickup and delivery locations (latitude, longitude), and delivery deadlines.
- Delivery personnel are associated with specific dark stores, and clustering is performed periodically, e.g., every 5 minutes.
- The system must handle high volumes during peak times, with orders potentially ranging from hundreds to thousands per hour per dark store.

System Architecture and Components

The architecture is designed using AWS services for cloud deployment, ensuring scalability and reliability. The key components are:

1. **Order Ingestion API**:
    - Receives new orders from the company's ordering system via HTTP requests.
    - Validates data and stores it in DynamoDB with status "pending".
    - **Technology**: AWS API Gateway for handling concurrent requests.
2. **Database**:
    - Stores order and cluster data with high throughput and scalability.
    - **Orders Table Schema**:

Field	Type	Description
order_id	String	Unique identifier for the order
dark_store_id	String	ID of the serving dark store
pickup_latitude	Number	Latitude of pickup location
pickup_longitude	Number	Longitude of pickup location
delivery_latitude	Number	Latitude of delivery location
delivery_longitude	Number	Longitude of delivery location
delivery_deadline	Number	Timestamp for delivery deadline
status	String	Order status (pending, assigned, etc.)

Clusters Table Schema:

Field	Type	Description

cluster_id	String	Unique identifier for the cluster
dark_store_id	String	ID of the serving dark store
order_ids	List	List of order IDs in the cluster
creation_timestamp	Number	Timestamp when cluster was created

3. **Technology**: Amazon DynamoDB for scalable, NoSQL storage, using on-demand capacity for variable loads.

Clustering Scheduler:

- Triggers the clustering process periodically, e.g., every 5 minutes, to ensure timely updates.
- Technology: AWS CloudWatch Events for scheduling, ensuring reliability.

Clustering Module:

- A serverless function that processes pending orders:
  - Queries DynamoDB for orders with status = "pending" and delivery_deadline <= now + 30 minutes.
  - Groups orders by dark_store_id.
  - Applies DBSCAN clustering on delivery locations (latitude, longitude) for each group.
  - Creates clusters, including single-order clusters for noise points (unclustered orders).
  - Stores results in the Clusters table.
- **Technology**: AWS Lambda with Python, using scikit-learn for DBSCAN implementation.
- **Parameters**: eps (e.g., 0.5 km for maximum distance) and min_samples (e.g., 2 for minimum points) are configurable per dark store based on density.

Cluster Retrieval API:

- Provides an endpoint for the order allocation system to retrieve current clusters.
- Queries DynamoDB for clusters with recent creation_timestamp (e.g., within last 5 minutes).
- Technology: AWS API Gateway for HTTP access.

Data Flow and Processing

The data flow is as follows:

1. **Order Ingestion**: New orders are sent to the Order Ingestion API, validated, and stored in DynamoDB.
2. **Clustering Process**: Every 5 minutes, CloudWatch Events triggers the Clustering Lambda, which:
   - Queries DynamoDB for eligible pending orders.
   - Groups by dark store and applies DBSCAN clustering.

    - Stores clusters in DynamoDB, ensuring all orders (including noise) are part of a cluster.
3. **Cluster Retrieval**: The order allocation system queries the Cluster Retrieval API to get recent clusters for assignment.

Clustering Algorithm and Model Selection

- **Algorithm**: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is selected for its ability to handle geospatial data with varying densities and noise, making it ideal for clustering delivery locations.
- **Temporal Consideration**: Orders are filtered by delivery_deadline <= now + 30 minutes before clustering, ensuring temporal relevance.
- **Handling Noise**: Noise points (unclustered orders) are assigned to single-order clusters, ensuring comprehensive coverage for allocation.
- **Parameters**: eps and min_samples are tuned based on the geographical density of each dark store's area, with typical values like 0.5 km and 2, respectively.

Cloud Deployment Architecture

The system leverages AWS for cloud deployment, ensuring scalability and cost-effectiveness:

- **Services Used**:
    - AWS API Gateway for ingestion and retrieval APIs.
    - Amazon DynamoDB for storing orders and clusters, with on-demand capacity for scalability.
    - AWS Lambda for serverless clustering, scaling automatically based on load.
    - AWS CloudWatch Events for scheduling, ensuring periodic execution.
- **Scalability**: Lambda scales with concurrent invocations, and DynamoDB handles variable read/write throughput, suitable for peak loads during festivals or sports events.
- **Cost Feasibility**: Pay-per-use model with Lambda and DynamoDB on-demand minimizes costs, with no upfront provisioning.
- **Fault Tolerance**: DynamoDB replicates across multiple Availability Zones, and Lambda ensures high availability with automatic retries.
- **Monitoring**: AWS CloudWatch monitors Lambda execution time, DynamoDB throughput, and API latency, ensuring operational visibility.

Practical and Technical Considerations

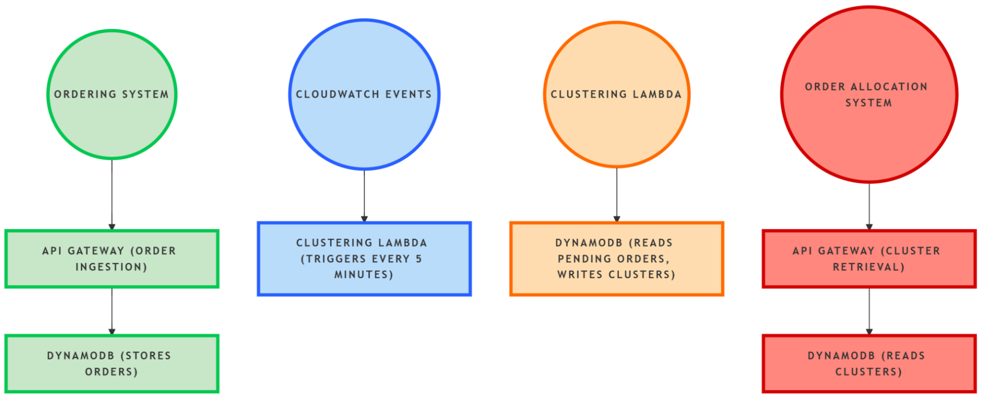
- **System Load**: The system handles bursts during peak hours, with DynamoDB and Lambda scaling automatically. For example, during festivals, order volumes may increase to thousands per hour, managed by on-demand capacity.
- **Concurrency**: API Gateway and Lambda handle concurrent order placements and clustering triggers, ensuring no bottlenecks.
- **Compute Costs**: Lambda is cost-effective for variable workloads, with clustering being lightweight (e.g., DBSCAN on a few hundred points per dark store).

- **Database Scaling**: DynamoDB scales horizontally, with on-demand capacity handling increased load without manual intervention.
- **Real-time Processing**: Periodic clustering every 5 minutes aligns with quick commerce timelines (10-30 minute deliveries), balancing computational cost and timeliness.
- **Outlier Events**: During high-demand events (e.g., cricket matches), the system scales automatically, with potential adjustments to clustering frequency (e.g., every 1 minute) if needed.
- **Operational Feasibility**: The design ensures minimal latency for order processing, with clustering results available for immediate use by the allocation system.

Functional Block Diagram

Due to text limitations, the functional block diagram is described as follows:

- **Components**: Ordering System, API Gateway, DynamoDB, CloudWatch Events, Clustering Lambda, Order Allocation System.


ORDERING SYSTEM
API GATEWAY (ORDER INGESTION)
DYNAMODB (STORES ORDERS)
CLOUDWATCH EVENTS
CLUSTERING LAMBDA (TRIGGERS EVERY 5 MINUTES)
CLUSTERING LAMBDA
DYNAMODB (READS PENDING ORDERS, WRITES CLUSTERS)
ORDER ALLOCATION SYSTEM
API GATEWAY (CLUSTER RETRIEVAL)
DYNAMODB (READS CLUSTERS)

Notes on Order Metadata

- **Required Information**:
  - dark_store_id: Identifies the serving dark store.
  - pickup_location: Latitude and longitude of the pickup point.
  - delivery_location: Latitude and longitude of the delivery point.
  - delivery_deadline: Timestamp for delivery deadline.
- **Collection**:
  - dark_store_id: Determined based on customer's delivery address or location.
  - pickup_location: Fixed for each dark store.
  - delivery_location: Geocoded from customer's address during order placement.
  - delivery_deadline: Set based on promised delivery time (e.g., 30 minutes from order placement).
- **Processing**:

    - Orders are stored in DynamoDB upon placement, with status updated as needed.
    - Clustering considers delivery_location and delivery_deadline for grouping.

Conclusion

This system design provides a scalable, cloud-based solution for clustering orders in a quick commerce environment. By leveraging AWS services, the system ensures high availability, cost efficiency, and the ability to handle variable loads. The use of **DBSCAN** for geospatial clustering, combined with periodic processing, balances real-time requirements with computational feasibility. The design addresses practical considerations like peak load handling, concurrency, and operational feasibility, making it suitable for production deployment.