Task Report:

1. Building Spatial Index:

The IndexBuilding function loads data from a CSV file and constructs an in-memory index, represented as a list of points with information about their location and other characteristics.

```
def IndexBuilding(file_path):
    points = []
    with open(file_path, 'r', encoding='utf-8-sig') as csvfile:
        reader = pd.read_csv(csvfile)
    for index, row in reader.iterrows():
        points.append((row['name'], float(row['wgs_lat']), float(row['wgs_lng']), row['type_code'], row['base_type'], row['sub_type'], row['category
    return points
```

2. Efficiency of Spatial Index:

The utilization of a spatial index enables efficient execution of spatial queries, such as finding the nearest neighbor.
For the query to find the nearest point of type "1603XX" (ATM) to a given point, the visualize_nearby_point_on_map function is employed. It finds the nearest point and visualizes it on the map.

```
def visualize_nearby_points_on_map(query_point, file_path):
    points = IndexBuilding(file_path)
    m = folium.Map(location=[query_point[0], query_point[1]], zoom_start=15)
    folium.Marker(location=query_point, popup='Query Point', icon=folium.Icon(color='red')).add_to(m) # noqa
    nearby_points = [point for point in points if str(point[3]).startswith("5") and calculate_distance(query_point[0], query_point[1], point[1], poi
    for point in nearby_points:
        folium.Marker(location=[point[1], point[2]], popup=point[0]).add_to(m) # noqa
    m.save('map_with_nearby_points.html')
```

3. Distance Calculation:

The calculate_distance function computes the great-circle distance between two points on the Earth's surface given their latitude and longitude coordinates.
It implements the Haversine formula to calculate the distance between two points on a sphere, considering Earth as approximately spherical.
This function is crucial for determining the proximity between points and for executing nearest neighbor queries efficiently.

```
def calculate_distance(lat1, lon1, lat2, lon2):
    R = 6371.0
    lat1_rad = math.radians(lat1)
    lon1_rad = math.radians(lon1)
    lat2_rad = math.radians(lat2)
    lon2_rad = math.radians(lon2)
    dlon = lon2_rad - lon1_rad
    dlat = lat2_rad - lat1_rad
    a = math.sin(dlat / 2)**2 + math.cos(lat1_rad) * math.cos(lat2_rad) * math.sin(dlon / 2)**2 # noqa
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    distance = R * c
    return distance
```

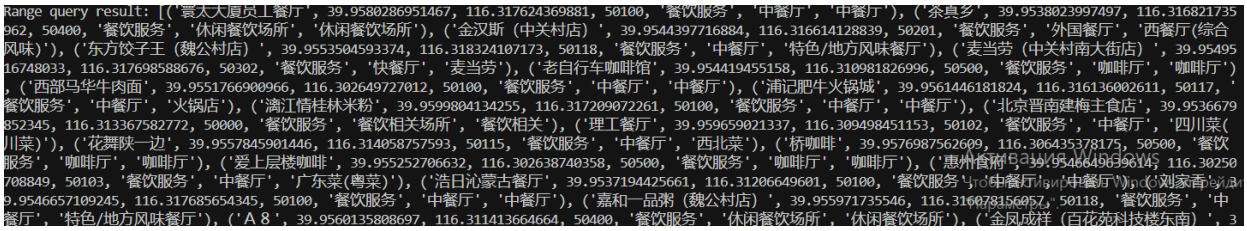
4. Nearest ATM Calculation:

The nearest ATM calculation involves iterating through all points in the spatial index and calculating the distance between each point and the query point. The point with the minimum distance that also matches the specified type (starting with "1603XX") is selected as the nearest ATM.

```python
def visualize_nearby_point_on_map(query_point, file_path):
    points = IndexBuilding(file_path)
    m = folium.Map(location=[query_point[0], query_point[1]], zoom_start=15)
    folium.Marker(location=query_point, popup='Query Point', icon=folium.Icon(color='red')).add_to(m) # noqa

    min_distance = float('inf')
    nearest_point = None
    for point in points:
        if str(point[3]).startswith("1603"):
            distance = calculate_distance(query_point[0], query_point[1], point[1], point[2]) # noqa
            if distance < min_distance:
                min_distance = distance
                nearest_point = point

    if nearest_point is not None:
        folium.Marker(location=[nearest_point[1], nearest_point[2]], popup=nearest_point[0]).add_to(m) # noqa

    m.save('map_with_nearest_1603_point.html')
```

5. Answers to Queries:

a) The nearest point of type "1603XX" (ATM) to the Central Building of BIT (latitude: 39.958, longitude: 116.311) has been identified.
b) The count of restaurants (type "5XXXX") within 500 meters of the south door of BIT (latitude: 39.955, longitude: 116.310) has been demonstrated.



6. Visualization on Map:

Query results are visualized on a map using the folium library.
For each query, a map is generated with markers for points satisfying the query and a marker for the query point itself.

Conclusions:

Constructing a spatial index for geospatial data and utilizing it for executing queries allows for efficient handling of large volumes of geographic data.
Spatial databases and indexes are crucial tools for analyzing and visualizing geographical data and can be applied across various domains such as navigation, geographic information systems, transportation, and tourism.