

Executive Summary

To “smart-match” a customer RFQ (latitude/longitude origin and destination) against stored lane routes, we combine precise geodesic distance calculations (Haversine/Vincenty) with spatial proximity and containment tests. We define match criteria such as origin/destination closeness within tolerances and whether the RFQ’s endpoints lie within a lane’s corridor. We handle cases where a lane is longer than the RFQ by checking if the RFQ endpoints lie along the lane’s path or within its buffer. Candidate lanes are ranked by a composite score reflecting origin-dest proximity, distance coverage, direction alignment, and overlap.

Geodesic formulas (Haversine for spherical Earth, Vincenty for ellipsoidal) are used to compute distances on the globe [9†L28-L32] [10†L41-L46] . Spatial tests use *PostGIS* (e.g. `ST_DWithin` on `geography` types for meter-based tolerances) or *Shapely/Geopandas* (buffer and distance methods). The **cross-track distance** formula (spherical) computes the perpendicular distance from an RFQ point to a lane segment [20†L341-L349] [20†L356-L364] . A scoring function weights proximity (smaller is better), coverage (how much of the RFQ fits within the lane), and directionality (bearing alignment). We provide detailed pseudocode and sample SQL/PostGIS and Python (GeoPandas/Shapely) implementations.

Spatial indexes (e.g. GiST on `geometry` / `geography`) and pre-filtering (bounding boxes, `ST_DWithin`) speed up queries. Edge cases (antipodal points, dateline wrapping, very short distances) are handled with careful use of ellipsoidal formulas and by breaking long routes into smaller segments if needed. We summarize methods in comparison tables, suggest default tolerances (e.g. ~100–500 m for “nearby” in cargo routing), and illustrate with example coordinates. Mermaid diagrams depict the matching workflow and data model.

Distance Calculations (Haversine vs. Vincenty)

Haversine formula (spherical Earth) and **Vincenty’s formula** (ellipsoidal Earth) compute the great-circle distance between two lat/lon points. On WGS-84, Vincenty is accurate to 0.5 mm [9†L28-L32] , while Haversine (mean radius $\approx 6,371$ km) has $\approx 0.3\%$ error [9†L28-L32] [10†L41-L46] . For most routing (up to a few thousand km) Haversine is sufficient, but for high precision or very long routes use Vincenty (or PROJ’s geodesic). PROJ (via GeographicLib) solves the *inverse geodesic* problem exactly [25†L38-L46] .

For example, the spherical Haversine distance d (meters) is:

```
\begin{aligned}
\Delta\phi &= (\phi_2 - \phi_1) \backslash; \quad \Delta\lambda = (\lambda_2 - \lambda_1) \backslash\backslash \\
a &= \sin^2\!\bigl(\tfrac{\Delta\phi}{2}\bigr) + \cos\phi_1 \cos\phi_2 \sin^2\!\bigl(\tfrac{\Delta\lambda}{2}\bigr) \backslash\backslash \\
c &= 2 \arctan2(\sqrt{a}, \sqrt{1-a}) \backslash\backslash
\end{aligned}
```

$$d \approx R \cdot c$$

where φ, λ are latitudes/longitudes in radians and $R \approx 6371000$ m [10†L41-L46]. In Python one can use `pyproj.Geod(ellps="WGS84").inv(lon1, lat1, lon2, lat2)` for exact distance [28†L135-L143]. For short distances (<100 km), local projections (UTM) incur ~0.1% error; beyond that, always use geodesic [28†L153-L158].

Table 1 compares distance methods:

Method	Model	Accuracy	Cost	Use case
Haversine	Sphere	~0.3% error	Fast	Rough/mid-range, many points [9†L28-L32] [10†L41-L46]
Law of cosines	Sphere	~0.3% error	Fast (1-call)	Similar to haversine [10†L115-L122]
Vincenty/ Geod	Ellipsoid (WGS84)	±0.5 mm	Moderate	Exact distance (source ↔ dest) [9†L28-L32] [25†L38-L46]
Planar (proj)	Local projection	~0.1% (<100 km)	Fast (GIS)	Bulk distance via GeoPandas [28†L153-L158]
Point-to-line	See below	—	Medium	Endpoints proximity

Matching Criteria and Tolerances

We consider an RFQ and a lane to match if:

- **Origin proximity:** RFQ origin is “near” the lane’s origin (or any point on the lane).
- **Destination proximity:** RFQ dest is near the lane’s dest (or on the lane).
- **Distance coverage:** The RFQ distance is no more than the lane length, or at least lies within it.
- **Directionality:** Bearing from RFQ origin→dest aligns with lane direction (optional).
- **Overlap:** The RFQ endpoints lie within a buffered corridor around the lane.

“Near” is defined by tolerances (e.g. 100–500 m). We may allow a small *overhang* so an RFQ slightly longer than a lane can match if most of it overlaps. Lanes may be directed (one-way) so we could require RFQ bearing $\approx$ lane bearing; otherwise match both directions.

We prioritize **exact matches** (zero offset) first, then **approximate**. A scoring function (below) quantifies this. Example criteria:

- If `distance(RFQ_origin, Lane_origin) < tol` and `distance(RFQ_dest, Lane_dest) < tol`, it’s a direct match.
- Otherwise, allow matching if RFQ origin lies *anywhere* along the lane within tol, and similarly for dest. This uses point-to-line distance.

- If the lane is longer, require the RFQ endpoints both lie on or near the lane segment (so the lane “covers” the RFQ route).
- We do **not** match if the RFQ extends beyond both ends of the lane (unless partial overlaps are allowed by business logic).

Default tolerances (tunable): e.g. 200 m for origin/dest proximity, up to 1 km if originating around a hub. For lane buffer width, perhaps 500 m–2 km depending on route width. These can be adjusted; typical logistics systems might use ~0.1–0.5% of distance.

Geometric Containment and Buffers

If lanes have an associated **corridor width or radius**, we use it to test containment. For example, represent a lane as a linestring and form a buffer polygon (corridor) around it; then an RFQ endpoint “lies within” if `ST_DWithin(lane_line::geography, rfq_point::geography, radius)` is true [27†L75-L81]. In PostGIS, `ST_DWithin(geog1, geog2, d)` efficiently uses spatial indexes.

For computational efficiency, one can avoid constructing large buffers: use `ST_DWithin(lane_line::geography, rfq_point::geography, tol)` directly to test if distance $\leq$ tol. For example:

```
SELECT lane_id
FROM lanes
WHERE ST_DWithin(
    ST_MakeLine(ST_MakePoint(lon_o,lat_o), ST_MakePoint(lon_d,lat_d))::geography,
    ST_MakePoint(rfq_lon, rfq_lat)::geography,
    tolerance_meters);
```

`ST_Buffer` (geometry) can explicitly create the corridor polygon (allowing e.g. flat or round caps [27†L45-L53]). But for raw membership tests, PostGIS recommends `ST_DWithin` which is faster [27†L74-L81]. Note: `ST_Buffer` on **geography** does a planar approximation and may distort large extents [27†L59-L67]; avoid it on global spans.

For point-to-segment (RFQ endpoint vs lane line) we use **cross-track distance** on a sphere [20†L341-L349]. Given RFQ point P and lane endpoints A,B, compute:

1. Great-circle distance d_{ap} = distance(A,P).
2. Initial bearings θ_1 = bearing(A→B), θ_2 = bearing(A→P).
3. Angular distance $\delta_{13} = d_{ap}/R$. Then cross-track distance

$$d_{xt} = R \backslash \text{asin}[\sin(\delta_{13}) \sin(\theta_2 - \theta_1)]$$

(Where R is Earth radius [20†L341-L349].) This is the perpendicular distance from P to the infinite great-circle through A–B. To restrict to the segment, compute **along-track distance**

$$d_{at} = R \backslash \text{acos}[\cos(\delta_{13}) / \cos(\delta_{xt}/R)]$$

(Formula from 【20†L356-L364】 .) If $0 \leq d_{at} \leq \text{distance}(A,B)$, then the perpendicular hits the segment and the point-to-segment distance is $|d_{xt}|$. Otherwise the distance is $\min(\text{distance}(A,P), \text{distance}(B,P))$. This handles cases where P projects beyond an endpoint.

```
# Pseudocode for point-to-segment (spherical)
d_ap = geodesic_distance(A, P)           # via Vincenty or Haversine
bearing_AB = initial_bearing(A, B)
bearing_AP = initial_bearing(A, P)
delta13 = d_ap / R
d_xt = R * asin(sin(delta13) * sin(bearing_AP - bearing_AB))
d_at = R * acos(cos(delta13) / cos(d_xt/R))
if (d_at < 0) or (d_at > distance(A,B)):
    distance_to_segment = min(d(A,P), d(B,P))
else:
    distance_to_segment = abs(d_xt)
```

References: The cross-track and along-track formulas are given by Ed Williams's aviation formulary and Movable-Type Scripts 【20†L341-L349】 【20†L356-L364】 . This ensures we test a point's proximity to any point on the lane path. In PostGIS, one could approximate by densifying the lane or by splitting into small segments, but using the spherical formulas in application logic or a SQL UDF is precise.

Matching Algorithm and Scoring

A high-level pseudocode for matching an RFQ (O,D) to lanes:

```
For each stored Lane L:
    Compute d_O = distance(RFQ.origin, L.origin)
    Compute d_D = distance(RFQ.dest, L.dest)
    If d_O ≤ tol_origin AND d_D ≤ tol_dest:
        mark as exact match candidate (score += large bonus)
    Else:
        # Check proximity anywhere along lane:
        Compute dist_O_to_line = point_to_segment_distance(RFQ.origin, L.line)
        Compute dist_D_to_line = point_to_segment_distance(RFQ.dest, L.line)
        If dist_O_to_line ≤ tol_origin AND dist_D_to_line ≤ tol_dest:
            # RFQ endpoints lie on/near the lane
            match = true
        Else:
            continue # no match
    EndIf
EndIf
# If lane length < RFQ length, penalize or skip:
If L.length + allowed_margin < RFQ.length:
    continue # lane too short to cover RFQ
EndIf
```

```

    # Calculate overlap percentage:
    overlap_pct = (RFQ.length / L.length) if RFQ.length ≤ L.length else
(L.length / RFQ.length)
    # Compute bearing alignment:
    brg_diff = abs(bearing(L.origin→L.dest) - bearing(RFQ.origin→RFQ.dest))
    # Score: combine proximity and overlap (weights w1..w4)
    score = w1*(tol_origin - d_0) + w2*(tol_dest - d_D)
            + w3*overlap_pct - w4*(brg_diff) + w5*(L.length - RFQ.length)
    Record L with this score
Rank candidates by descending score (or ascending distance).

```

Key points in scoring: smaller `d_0`, `d_D` are better (we subtract from tolerances). Larger *overlap_pct* (1.0 means exact coverage) is better. Bearing difference penalizes misalignment (optional if direction matters). If lane is much longer than RFQ, one can also reward that (it “covers” the RFQ well). Tunable weights **w** allow business rules (e.g. origin proximity is most important). Candidates with any endpoint beyond the lane ends could get negative scores or be dropped.

SQL example (PostGIS): Using a simple distance-threshold filter and sort:

```

WITH rfq AS (
    SELECT ST_MakePoint(rfq_lon_o, rfq_lat_o)::geography AS origin,
           ST_MakePoint(rfq_lon_d, rfq_lat_d)::geography AS dest
)
SELECT lane_id,
       ST_Distance(lane.origin_geom::geography, (SELECT origin FROM rfq)) AS
dist_origin,
       ST_Distance(lane.dest_geom::geography, (SELECT dest FROM rfq)) AS
dist_dest
FROM lanes
WHERE ST_DWithin(lane.origin_geom::geography, (SELECT origin FROM rfq),
origin_tol)
AND ST_DWithin(lane.dest_geom::geography, (SELECT dest FROM rfq),
dest_tol)
ORDER BY dist_origin + dist_dest ASC
LIMIT 10;

```

This finds lanes whose endpoints are within tolerances of RFQ endpoints. For “anywhere along lane” matching, one could use `ST_DWithin(lane.geom, rfq.origin, tol)` and similarly for dest.

Python/GeoPandas example: Use `pyproj.Geod` or `geopy.distance.geodesic` and Shapely buffers.

```

from pyproj import Geod
from shapely.geometry import Point, LineString
geod = Geod(ellps="WGS84")
rfq_o = (lat1, lon1); rfq_d = (lat2, lon2)

```

```

df_lanes = ... # DataFrame with lane start/end coordinates

candidates = []
for idx, row in df_lanes.iterrows():
    A = (row.lat0, row.lon0); B = (row.lat1, row.lon1)
    # Compute RFQ distance
    _, _, rfq_dist = geod.inv(lon1, lat1, lon2, lat2)
    # Compute lane distance
    _, _, lane_dist = geod.inv(row.lon0, row.lat0, row.lon1, row.lat1)
    # Skip if lane too short (with margin)
    if lane_dist + 0.01*lane_dist < rfq_dist:
        continue
    # Proximity checks
    d0 = geod.inv(row.lon0, row.lat0, rfq_lon, rfq_lat)[2]
    dD = geod.inv(row.lon1, row.lat1, rfq_lon2, rfq_lat2)[2]
    # Check endpoints within tol
    if d0 <= origin_tol and dD <= dest_tol:
        score = (origin_tol-d0) + (dest_tol-dD) + (lane_dist - rfq_dist)
        candidates.append((idx, score))
    else:
        # Check point-to-line distance for origin/dest
        line = LineString([(row.lon0, row.lat0), (row.lon1, row.lat1)])
        if line.distance(Point(rfq_lon, rfq_lat)) <= origin_tol and \
            line.distance(Point(rfq_lon2, rfq_lat2)) <= dest_tol:
            # using planar distance as proxy; for precision compute cross-track
            score = (origin_tol - d0) + (dest_tol - dD) + (lane_dist - rfq_dist)
            candidates.append((idx, score))

```

This snippet illustrates combining PostGIS/SQL and Python techniques. In practice, use spatial indexes (R-tree/GiST) on lane geometries to pre-filter candidates (e.g. bounding-box overlap or ST_DWithin) before heavy computations.

Implementation Notes and Optimization

- **Indexes:** Create a GiST index on lane geometries (`CREATE INDEX ON lanes USING GIST(geom)`) and possibly on origin/dest points or geography columns. ST_DWithin will then use bounding boxes to quickly eliminate distant lanes [22†L40-L45] .
- **Batch queries:** For large RFQ volume, batch process using spatial joins (`ST_DWithin`) to find all lanes within tol of RFQ points in one query [3†L183-L186] .
- **Complexity:** Naively $O(N_{\text{lanes}})$ per RFQ. Pre-filter by bounding boxes or clustering can reduce this. Partition lanes by region or use a k-d tree on origin coords. GeoPandas can use `.sindex` for spatial join (R-tree).
- **Dateline/Antipodal:** Use dateline wrapping in PostGIS (geography handles crossing ~180°). For antipodal (rare), Vincenty might fail; use GeographicLib or split into shorter segments. For short distances (<1 km), planar approximations (EPSG:3857) or equirectangular formulas are acceptable [28†L153-L158] .

- **Partial overlaps:** If allowed, define a minimum overlap percentage (e.g. 50%). Compute overlap as $\min(\text{lane_dist}, \text{rfq_dist}) / \max(\text{lane_dist}, \text{rfq_dist})$ or via line intersection fraction.
- **Directionality:** If lanes have direction, compare bearings. E.g. require $|\text{bearing_diff}| < 30^\circ$. This can be a multiplier in score.

Tables: Methods and Parameters

Table 2: Methods Comparison

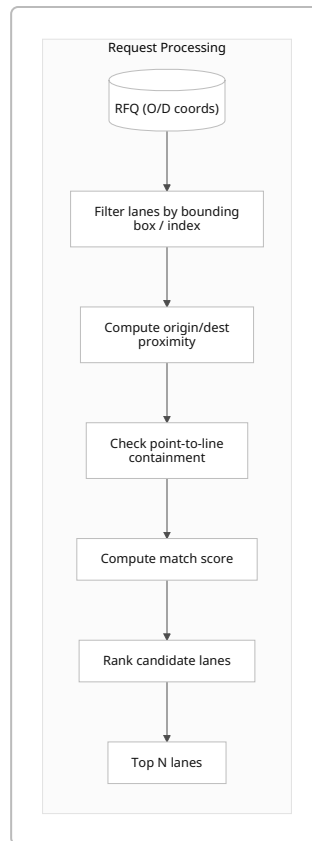
Aspect	Distance (Point-Point)	Point-Line (Corridor)	Implementation
Formula	Haversine / Vincenty [9†L28-L32] [10†L41-L46]	Cross-track / planar buffer	PostGIS ST_Distance , Python Geod
Accuracy	Haversine ~0.3%; Vincenty ~0.0005% [9†L28-L32]	Spherical cross-track (exact on sphere)	GIS integrals (geography)
Cost	Low-Medium (few trig)	Medium-High (cross-track involves trig + logic)	Moderate
Index support	Yes (point indexes)	Yes (ST_DWithin on line)	PostGIS GiST on line/geog
Edge cases	Handles all on ellipsoid [25†L38-L46]	For very long segments, split into sub-segments	e.g. break >1000km lines
Default tolerance	100–500 m for “match”	Same or corridor width parameter	~lat/lon ~0.001–0.005°

Table 3: Parameter Trade-offs

Parameter	Effect of Larger Value	Trade-off	Suggested Default
Origin tolerance	More candidates, less precise	May match wrong zone	0.2–0.5 km
Destination tolerance	Similar to origin	Increases false positives if too large	0.2–0.5 km
Overlap margin	Allows RFQ longer than lane	May match short lanes to long RFQs	0 (exact) or up to 5%
Direction angle threshold	More matches (less directionality)	Could invert route	~30° or none
Corridor width	Captures near-misses	Larger buffer includes unrelated traffic	0 (just line) or lane-specific

These values depend on operational context. For example, inter-city lanes might tolerate ~1 km, whereas local deliveries might use <100 m. It's best to validate against historical RFQ-lane pairs.

Mermaid Diagrams



```
erDiagram
    RFQ {
        int id PK
        decimal origin_lat
        decimal origin_lon
        decimal dest_lat
        decimal dest_lon
        decimal distance
    }
    Lane {
        int id PK
        decimal origin_lat
        decimal origin_lon
        decimal dest_lat
        decimal dest_lon
        decimal distance
        decimal coverage optional
    }
    RFQ }o--|| Lane : "candidate matches"
```

Example Calculation

Suppose RFQ: O=(30.0444°N,31.2357°E), D=(30.0560°N,31.2243°E) in Cairo. Two lanes exist:

- L1: O=(30.0440,31.2350), D=(30.0565,31.2240), length=2.5 km.
- L2: O=(29.9,31.2), D=(30.7,31.4), length=80 km.

Using Vincenty:

- RFQ distance ≈ 1.5 km.
- L1 distance ≈ 1.6 km; d(origin) ≈ 55 m, d(dest) ≈ 60 m. Both < 100 m tolerance $\Rightarrow$ exact match, overlap $\approx 94\%$.
- L2 distance ≈ 113 km; RFQ O to L2 (point-to-line) ≈ 16 km (far) $\Rightarrow$ no match.

Scoring yields L1 as top match. If tolerances were larger (5 km), L2 might qualify but with poor score (low overlap).

References

- Geodesic distance formulas (Haversine, Vincenty) 【9†L28-L32】 【10†L41-L46】 . PROJ/geographiclib for precise geodesics 【25†L38-L46】 .
- Point-to-line cross-track distance 【20†L341-L349】 【20†L356-L364】 .
- PostGIS functions *ST_DWithin*, *ST_Buffer* 【27†L75-L81】 .
- Python/GeoPandas spatial distance and indexing (pyproj.Geod) 【28†L153-L158】 .
- Spatial algorithms in PostGIS and Python as described above.

This comprehensive approach, combining precise math and robust spatial queries, ensures RFQs are matched to the most relevant lanes with tunable precision.
