A Comparative Evaluation of Spatio-Temporal Indexing Strategies on Large-Scale Trajectory Data

Xiaoyu Zhuang

***Abstract*—With the rapid development of artificial intelligence, large-scale taxi trajectory data has been leveraged everywhere. This study benchmarks hybrid indexing strategies on a large dataset to guide the selection of indices for diverse query scenarios in spatial databases.**

***Index Terms*—Spatio-Temporal Query, Skyline Query, Trajectory Similarity Analysis**

I. Introduction

With the rapid development of artificial intelligence, large-scale taxi trajectory data has been leveraged in domains like urban planning, traffic management, and recommender systems. Such trajectory data fuel personalised route recommendation algorithms for drivers and passengers, and serves as training data to enhance autonomous driving system navigation and decision-making [1]. However, managing and querying massive spatio-temporal trajectory datasets poses significant performance challenges. They usually cause query performance to deteriorate as data volume and dimensionality grow [2]. These limitations motivate advanced indexing methods and algorithms to ensure query efficiency in spatial databases.

To explore how different indexing strategies affect query algorithms and performance, we utilise the Porto taxi dataset [3] from 2013 to 2014 with over 1.7 million GPS trajectories across the city. In addition, we also evaluate the performance of measuring trajectory similarity in high-dimensional mobility datasets. To find the best strategies for optimising queries in a spatial database, we design three different query scenarios: **Spatio-Temporal Query**, **Skyline Query** and **Trajectory Similarity Analysis**.

A. Spatio-Temporal Query

The Spatio-Temporal query aims to efficiently retrieve trajectories within specified spatial and temporal boundaries, meeting spatial coverage criteria within defined time intervals. Such queries have direct implications in traffic flow analysis, where both time and geography are crucial. In this experiment, we evaluate three indexing strategies: firstly, B-tree indexing for temporal attributes, effectively narrowing down data based on timestamps; secondly, GiST and SP-GiST indexing for handling spatial attributes, optimised for geographic constraints; and finally, a hybrid indexing approach combining B-tree with GiST or SP-GiST. This combined method leverages the temporal filtering strength alongside the spatial query capabilities to reduce query processing times significantly.

B. Skyline Query

The Skyline query is designed to retrieve trajectories that optimise trade-offs between travel time and distance. Such queries hold significant practical value in applications like route planning and decision support systems, where multiple competing objectives must be balanced simultaneously. In our experiment, we systematically evaluate the performance of different spatial indexing methods, including sequential scanning (baseline without indexing), GiST indexing and SP-GiST indexing. The rationale behind selecting GiST and SP-GiST indexing structures stems from their ability to efficiently handle spatial data types and rapidly prune search spaces based on geometric criteria, thus reducing computational overhead and improving query responsiveness.

C. Trajectory Similarity Analysis

The trajectory similarity query identifies the top five trajectories most similar to a given reference route by applying the Fast Dynamic Time Warping (FastDTW) algorithm. A custom-defined function (UDF) is implemented to filter candidate trajectories efficiently and different indexing strategies are also applied. This analysis investigates how FastDTW performance varies under polylines composed of different point numbers and indexing combinations. It can provide critical insights for applications such as route recommendation systems, anomaly detection in urban mobility, and performance optimisation in trajectory-based similarity searches.

II. Methodology

This section details the methodological approach of our experiment, including data preprocessing, region preset, query task designs, and the objects of the selection and implementation of indexing strategies.

A. Data Preprocessing

The original dataset was provided in CSV format and represented by nine attributes to describe taxi trips. Data was preprocessed using a Python function `data_preprocessor()`. The primary objective was to transform raw GPS traces and timestamps into formats compatible with PostGIS, while also enriching the dataset

with derived attributes to support indexing and filtering. Specifically, preprocessing focuses on the following attributes:

- **polyline** - To enable robust spatial indexing and geometric analyses within a PostGIS framework, the raw JSON-encoded polyline, originally a nested list of [longitude, latitude] coordinate pairs, is deserialised via `json.loads` and reformatted into a Well-Known Text (WKT) `LINESTRING(lon lat, lon lat, ...)`. Trajectories that yield an empty polyline or comprise fewer than two coordinate pairs are systematically excluded, as they fail to constitute valid LineString geometries.
- **timestamps** - The original `TIMESTAMP` field (Unix epoch seconds) was replaced by a derived array of ISO 8601-formatted UTC timestamps. Each timestamp aligns with a GPS coordinate from the `POLYLINE`, assuming a fixed 15-second interval between points. This conversion improves interpretability and supports future time series analysis.
- **total_travel_time** - The duration of each trip was computed as $(\texttt{point\_count} - 1) * 15$ in seconds.
- **start_local_time** - The trip's starting time was derived from the Unix timestamp and converted to local time in the *Lisbon* timezone in YYYY-MM-DD HH:MM:SS format.
- **end_local_time** - The trip's ending time that is similar to `start_local_time`, computed by $\texttt{total\_travel\_time} + \texttt{start\_local\_time}$.
- **hour_of_day** - The trip's start hour (0 – 23).
- **point_count** - The number of GPS points per trip.

B. Region Preset

To facilitate spatial containment and proximity queries in subsequent tasks, six representative regions in Porto were predefined using polygon geometries. These areas include transportation hubs, commercial zones, and tourist landmarks, which were delineated via `geojson.io` [4] and encoded in **WGS 84** format using `ST_GeomFromText()` functions.

C. Experimental Design

*1) **Spatio-Temporal Query**:* This task focuses on retrieving taxi trips based on both time and spatial constraints. Specifically, we examine trajectories during morning rush hours (7 am – 9 am) that interact with a predefined city region. Three spatial-temporal patterns are explored: (1) trips fully contained within a region, (2) trips end inside but start outside the region, and (3) trips start and end inside but partially traverse outside.

In the queries, spatial functions are used to express geometric relationships. `ST_StartPoint` and `ST_EndPoint` return the first and last point of a `LINESTRING` geometry respectively. `ST_CoveredBy` checks if the full `LINESTRING` lies within the region; `ST_Contains` ensures specific points are spatially enclosed.

To explore the query optimisation performance of indexing on different data types, i.e. spatial data and temporal data, we implement and compare three indexing strategies: (i) **B-tree** on `start_local_time` and `hour_of_day` to accelerate temporal filtering; (ii) **GiST** on `polyline` and **SP-GiST** on `ST_StartPoint`/`ST_EndPoint` to optimise for spatial relationships; and (iii) **Combined Indexing**. This comparison helps reveal whether combining temporal and spatial indexes yields synergistic performance gains over single-dimensional indexes.

*2) **Skyline Query**:* This query aims to search optimal trajectories between a given origin and destination, where no other trip is strictly better in both travel time and route length. Known as the "skyline" set, such non-dominated trajectories are critical in multi-objective decision-making scenarios such as route optimisation in navigation.

The query is implemented using a SQL Common Table Expression (CTE) to filter relevant trips based on spatial relationships using `ST_Contains` on both `ST_StartPoint(polyline)` and `ST_EndPoint(polyline)`. The skyline logic is enforced using a self-join with `NOT EXISTS` to identify trajectories not dominated in both travel time and trip mileage.

In theory, PostGIS offers two major spatial indexing methods: **GiST** (Generalised Search Tree) and **SP-GiST** (Space-Partitioned GiST). GiST is widely used for spatial range queries due to its balanced-tree structure and flexibility with irregular or overlapping geometries [5]. It is the default index type for spatial operations like `ST_Contains` in PostgreSQL. In contrast, SP-GiST on the same fields uses space-partitioning (e.g. a quadtree), yielding a high fan-out that can accelerate searches on point geometries when data is uniformly distributed and non-overlapping [6]. This is common in start/end point distributions. However, the PostgreSQL query planner does not always prefer GiST or SP-GiST unless it sees a clear cost benefit. It will default to a full table scan if it expects an index to offer little benefit [7]. To evaluate these trade-offs in a skyline context, three indexing strategies were tested in skyline queries: (i) **Sequential Scan** (no index) as a baseline; (ii) **GiST** index on `ST_StartPoint` and `ST_EndPoint`, and (iii) **SP-GiST** index on the same fields.

*3) **Trajectory Similarity Analysis**:* This task aims to calculate the top 5 most similar trajectories to a given reference trip using the **Fast Dynamic Time Warping (FastDTW)** algorithm. Such similarity measures are valuable in conditions where geometric shape matters more than exact time alignment, such as route recommendation, pattern recognition and trajectory clustering.

To implement the FastDTW algorithm, we use `plpython3u` to define a user-defined function (UDF) `find_top5_similar_trips()` to encapsulate the entire query logic. The function takes a reference `trip_id` and computes the DTW distance between its trajectory and candidate trips filtered by spatial proximity, temporal overlap, and trajectory length similarity. The function returns the five closest matches based on DTW score.

To evaluate the performance of the Algorithm 1, we tested three index strategies: (i) **B-tree** on `trip_id`, `hour_of_day`, and `point_count` for non-spatial filters; (ii) **GiST** on `polyline`, `ST_StartPoint`, and `ST_EndPoint` to accelerate spatial filtering; and (iii) **Combined Indexing**, applying both types. This setup allows us

Algorithm 1: Find Top-5 Similar Taxi Trips by Fast-DTW

Input: Base trip ID b, count bounds $(\delta_{\min}, \delta_{\max})$
Output: Top-5 similar trips with smallest DTW distances

Retrieve base trip b's polyline, start and end point, hour, and point count;
Query candidate trips c_i where:
- trip $\neq b$
- start/end points within 500m of base trip
- similar hour of day
- point count in $[count_b + \delta_{\min}, count_b + \delta_{\max}]$
- trip length longer than 1000m

foreach *candidate trip* c_i **do**
  Parse polyline to coordinate array;
  Compute DTW distance d_i between b and c_i;
  Record c_i with its metadata and d_i;
Sort all candidates by DTW distance d_i;
return top-5 trips with smallest d_i;

to assess how pattern differences and index-assisted filtering impact DTW-based trajectory similarity.

III. Experimental Results & Analysis

This section presents a comparative evaluation of three query tasks under different indexing strategies.

A. Spatio-Temporal Query

The experimental plans and the following table reveal a hierarchy of effects delivered by temporal indexes, spatial indexes, and their interaction.

TABLE I: Execution Time (ms) for Spatio-Temporal Queries

Test Case	B-tree	GiST & SP-GiST	Combined Indexing
Case 1.1	66.308	193.939	65.668
Case 1.2	33.375	538.338	29.160
Case 1.3	41.773	376.955	153.169

1) ***Case 1.1 - Whole trajectory inside the CBD****:* Filtering on a month-long morning peak yields high temporal selectivity: the B-tree bitmap alone finishes in 66 ms, with 6,899 shared-buffer hits and a negligible 0.34 ms planning cost. Adding the GiST line index hardly changes execution time (65.7 ms) but inflates the bitmap build to 5,620 extra buffer touches, indicating that the GiST contributes little discrimination after the narrow time window is applied. Conversely, a spatial-only plan must first probe the GiST, then re-check every candidate with `ST_CoveredBy`, so it scans 8,561 pages and stalls at 194 ms. Therefore, when temporal granularity already restricts the search space, the GiST primarily adds CPU overhead, thus leading to a longer query time.

2) ***Case 1.2 - Start outside but end in CBD****:* Here the SP-GiST point index is highly selective: combined bitmap-ANDing (endpoint SP-GiST + two B-trees) yields the fastest result (29 ms) while reading only 2,656 buffers. Temporal-only pruning is still respectable (33 ms, 399 buffers), but leaves every row to evaluate two geometric predicates. Relying on the SP-GiST alone is disastrous: the index scan returns 97,000 candidate rows, driving buffer traffic above 98,000 and total time to 538 ms. Hence, when a point predicate provides strong spatial selectivity, pairing it with temporal bitmaps minimises both I/O and CPU.

3) ***Case 1.3 - Trajectory partly outside the CBD****:* `NOT ST_CoveredBy` is non-sargable for GiST, so the spatial bitmaps deliver almost no pruning. The temporal plan therefore wins at 41.8 ms with only 1 675 buffers read. A spatial-only strategy degrades to 153 ms and 34 k pages because every candidate polyline must be inspected geometrically. Worse, superimposing GiST and SP-GiST on the temporal bitmaps forces three lossy bitmaps to be merged and evaluated row by row, ballooning runtime to 377 ms and 43 k buffers. The combined plan's higher planning cost (0.35 ms versus 0.22 ms) is trivial, but its evaluation cost is dominant.

Overall, bitmap is advantageous only when every participating index is highly selective; adding a low-quality bitmap multiplies, rather than divides, the workload. In other words, more indexes are not necessarily better. Therefore, index design should privilege the dimension whose predicate is both highly selective and index-exploitable. Combined strategies are valuable only when each bitmap materially shrinks the candidate set; otherwise, simplicity outperforms sophistication.

B. Skyline Query

The experimental results are shown in the Table II.

1) ***Baseline****:* In the skyline experiment, executing the query without spatial indexes forced PostgreSQL to perform a parallel sequential scan followed by an anti-join on roughly 900 k trajectories, touching more than 265 k disk pages; the operation lasted about 0.35 s and incurred planning costs in the tens of millions because the optimiser possessed no selectivity statistics for the geometric predicates.

2) ***GiST****:* Adding a GiST composite index on `ST_StartPoint` and `ST_EndPoint` radically altered the plan: the optimiser substituted two bitmap index scans whose bitmaps were merged before a parallel heap probe. When start and end regions were both compact, such as "Airport to Station" or "Tourist to Stadium", the GiST R-tree exhibited minimal node overlap, so only a few dozen tuples survived to heap re-check and execution times fell to 4–20 ms with fewer than 500 shared-buffer hits. Under a near-uniform pick-up zone centred on the CBD, however, the same index had to traverse many sibling rectangles whose bounding boxes all intersected the query region. Consequently the bitmap expanded to 170 k candidates, prompting 7,650 page reads and stretching runtime to 112 ms. The planner's estimated cost was nevertheless identical to that in the clustered case because it assumes uniform point distributions and therefore cannot anticipate cross-page overlap within GiST subtrees.

3) ***SP-GiST****:* Replacing GiST with an SP-GiST point index retained the bitmap strategy but changed the underlying structure to a disjoint space-partitioned radix tree. In the CBD scenario, the partitioning eliminated rectangle overlap,

TABLE II: Comparison of GiST and SP-GiST in Skyline Query

Test Case	Execution time (ms)			Planning total cost			Shared buffers (hit/read)		
	Baseline	GiST	SP-GiST	Baseline	GiST	SP-GiST	Baseline	GiST	Mem.
Case 2.1	340.829	20.242	23.984	17891733.76	40.78	40.78	4512/265808	469/0	1607/0
Case 2.2	356.530	112.043	94.214	17891733.76	40.78	40.78	4744/265616	38/7650	3627/7545
Case 2.3	363.995	4.659	6.894	17891733.76	40.78	40.78	4670/265520	128/0	605/0

TABLE III: FastDWT Performance

Δpoint_count	Index	Candidate query (s)	Candidates number	DTW time (s)	Peak RAM (MB)	Total points
0–10	B-tree	0.159	157	0.421	4.34	7115
0–10	GiST	0.136	157	0.431	4.34	7115
0–10	Combined	0.150	157	0.439	4.34	7115
0–30	B-tree	0.229	8	0.025	0.29	467
0–30	GiST	0.009	8	0.036	0.29	467
0–30	Combined	0.022	8	0.032	0.29	467
No limit	B-tree	0.211	10	0.048	0.84	849
No limit	GiST	0.051	10	0.048	0.84	849
No limit	Combined	0.060	10	0.063	0.84	849

so fewer tuples were re-checked and execution shortened to 94 ms in spite of a similar number of page reads; this represents a 16% improvement over GiST. In clustered settings, the mandatory partition descents fragmented what would otherwise be contiguous leaf blocks, increasing heap re-checks (1.6 k) and raising elapsed time to 6–24 ms, which is slightly slower than GiST yet still two orders of magnitude faster than the baseline scan. Notably, the optimiser assigned an identical bitmap cost of 11.33 to both spatial indexes in every test, implying that its tie-breaking is effectively random when operator statistics are absent.

In conclusion, GiST excels whenever start or end points concentrate in spatial hot-spots, because its R-tree minimises overlap and maximises pruning efficiency, whereas SP-GiST is superior for uniformly distributed or highly mixed origin–destination patterns, where disjoint partitioning prevents the combinatorial explosion of overlapping rectangles that weakens GiST.

C. Trajectory Similarity Analysis

Table III shows that FastDTW's cost is governed almost exclusively by the amount of trajectory data, not by the choice of indexing method. Because the three index configurations deliver exactly the same candidate list for a fixed point count difference, their effect is limited to the candidate query phase. What drives the subsequent numeric kernel is the total number of points fed into FastDTW.

FastDTW achieves linear-time performance by coarse-graining each series into progressively lower-resolution layers and restricting local alignment to a fixed search radius r at every scale. With r constant, the theoretical time and space complexity are $\mathcal{O}(N)$ and $\mathcal{O}(N)$ respectively, which is different from the traditional DTW's $\mathcal{O}(N^2)$ [8]. When the procedure is invoked repeatedly, total cost is therefore proportional to the sum of point counts across all candidate pairs.

The data exhibit precisely that linear law. At Δpoint_count = 0 – 30 the eight candidates contain 467 vertices in total; FastDTW finishes in 25–36 ms and peaks at 0.29 MB, or roughly 0.62 kB per point. Removing the length filter (Δpoint_count = 0 – 10) allows 157 trajectories that together contribute 7,115 vertices. Runtime rises to 0.42 s—17.5 times higher—while memory climbs to 4.34 MB, a 15-fold increase. The intermediate "no-limit" setting (10 candidates, 849 points) yields 48 ms and 0.84 MB, both almost exactly proportional to the factor-of-1.8 jump in data volume relative to the narrow window. Small deviations from perfect linearity stem from Python interpreter overhead and the one-off allocation of the radius-bound warping window, which is more significant when N is small.

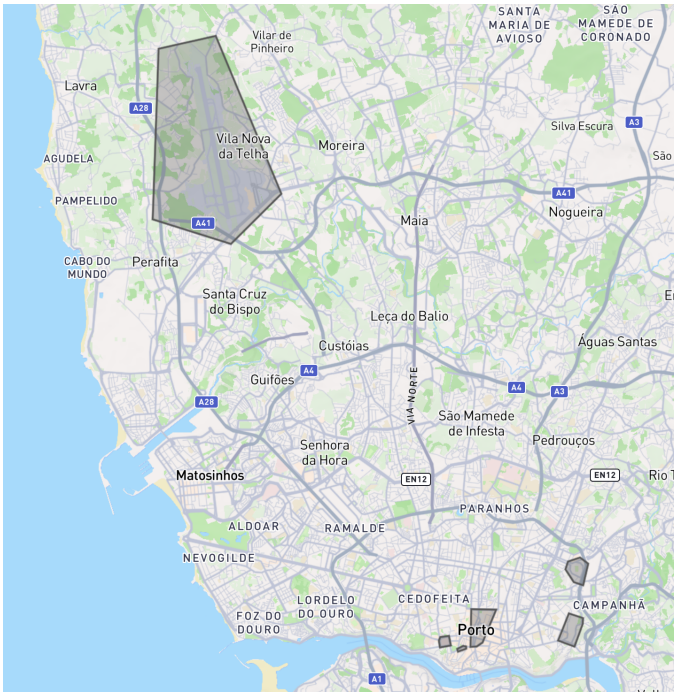
Because every candidate is matched against the same reference trip, the total vertex count equals the reference length plus the sum of candidate lengths. Pruning either dimension is beneficial: shortening the length window immediately lowers N, while reducing the number of candidates lowers the coefficient multiplying N. Indexes indirectly influence FastDTW by accelerating those predicates—hour, length, and spatial proximity—that suppress either dimension. Their direct impact on the numeric phase is nil.

In practical terms, scalability hinges on constraining point counts before the UDF executes. A tight point_count band or aggressive spatial filtering can keep N under a thousand and guarantee sub-50 ms similarity searches with sub-megabyte memory footprints. Conversely, admitting hundreds of trajectories that differ widely in sampling density multiplies both runtime and RAM linearly; no indexing strategy can offset that growth once the data have been loaded into Python. Therefore, any production pipeline that relies on DTW should first enforce coarse filters in SQL, then reserve FastDTW for a slim, length-controlled candidate set.

IV. Conclusion

This experiment has demonstrated indexing must match predicate selectivity. Combining indexes helps only when each bitmap prunes aggressively. Before costly analytics like FastDTW, enforce strict SQL filters to guarantee scalable performance and stability at the millisecond level.

Appendix A
Predefined Regions



Appendix B
Spatio-Temporal Query Template

```
SELECT trip_id, start_local_time,
    end_local_time, polyline
FROM taxi_trajectory
WHERE
  -- Temporal Data Filter
  start_local_time >= <date_and_time>
  AND start_local_time <  <date_and_time>
  AND hour_of_day BETWEEN <start_hour> AND <
      end_hour>
  -- Spatial Data Filter
  AND (
      -- Case (1): Trips fully contained
          within a region
      ST_CoveredBy(polyline, <region>)
    OR (
      -- Case (2): Trips that end inside but
          start outside the region
      ST_Contains(<region>, ST_EndPoint(
          polyline)) AND
      NOT ST_Contains(<region>, ST_StartPoint(
          polyline))
    )
    OR (
      -- Case (3): Trips that start and end
          inside but partially traverse
          outside
      ST_Contains(<region>, ST_StartPoint(
          polyline)) AND
      ST_Contains(<region>, ST_EndPoint(
          polyline)) AND
      NOT ST_CoveredBy(polyline, <region>)
    )
  );
```

Listing 1: Spatio-Temporal Query Template

Appendix C
Skyline Query Template

```
WITH filtered_trips AS (
  SELECT trip_id, total_travel_time, ST_Length
      (polyline) AS route_length, polyline
  FROM taxi_trajectory
  WHERE
    -- Filter the trips based on start and end
        regions
    ST_Contains(
      <region_1>, ST_StartPoint(polyline)
    ) AND
    ST_Contains(
      <region_2>, ST_EndPoint(polyline)
    )
)
SELECT *
FROM filtered_trips t1
WHERE NOT EXISTS (
  SELECT 1
  FROM filtered_trips t2
  WHERE
    -- Find skyline
    (t2.total_travel_time <= t1.
        total_travel_time AND
     t2.route_length <= t1.route_length) AND
    (t2.total_travel_time < t1.
        total_travel_time OR
     t2.route_length < t1.route_length)
);
```

Listing 2: Skyline Query Template

Acknowledgment

I acknowledge ChatGPT 4o is used for helping me check spelling consistency and correct grammar errors. Google Translate is used for translating complex sentences. These tools are only used to enhance the clarity and accuracy of parts of the content.

References

[1] X. Kong, Q. Chen, M. Hou, H. Wang, and F. Xia, "Mobility trajectory generation: a survey," *Artificial Intelligence Review*, vol. 56, no. Suppl 3, pp. 3057–3098, 2023.

[2] M. M. Alam, L. Torgo, and A. Bifet, "A survey on spatio-temporal data analytics systems," *ACM Comput. Surv.*, vol. 54, no. 10s, Nov. 2022. [Online]. Available: https://doi.org/10.1145/3507904

[3] C. Cross, "Taxi trajectory data," https://www.kaggle.com/datasets/crailtap/taxi-trajectory, accessed: 2025-05-01.

[4] "geojson.io," https://geojson.io/, accessed: 2025-05-05.

[5] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, *Generalised search trees for database systems*. September, 1995.

[6] W. G. Aref and I. F. Ilyas, "Sp-gist: An extensible database index for supporting space partitioning trees," *Journal of Intelligent Information Systems*, vol. 17, pp. 215–240, 2001.

[7] M. Christofides, "Some indexing best practices," https://www.pgmustard.com/blog/indexing-best-practices-postgresql, accessed: 2025-05-06.

[8] R. Wu and E. J. Keogh, "Fastdtw is approximate and generally slower than the algorithm it approximates," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 8, pp. 3779–3785, 2020.