MCTM: Multi-Chord Distributed System for Efficient Trajectory Data Management in Mobile Edge Computing

Yucheng Tao[1], Haopeng Chen[1] (✉), Zihong Lin[1], Jiahao Xu[1], Xiaojian Gao[1], Jinghao Wang[1], Yan Jiao[2], and Yongming Xu[2]

[1] Shanghai Jiao Tong University, Shanghai China
[2] Shanghai Shapere Information Technology Co., Ltd., Zhejiang, China
`{taoyucheng,chen-hp,923048992,ianhuiiii,gaoxiaojian,wjh2002@sjtu.edu.cn}`
`{jiao,xu}@shapere.xyz`

Abstract. Efficient management of large-scale trajectory data for mobile objects is a critical and challenging task. Existing systems struggle to handle the high volume of real-time trajectory messages in mobile edge scenarios, where high data transmission latency and overhead impede efficiency and real-time data management. Additionally, support for fine-grained representation and efficient query processing of trajectory data is often inadequate. This paper introduces MCTM, a multi-chord distributed system that efficiently stores and manages trajectory data on edge nodes, reducing network latency and overhead. MCTM presents a fine-grained temporal index (TI), an Index Cache, and a spatio-temporal index (TSI), significantly improving query efficiency and accuracy. In various query scenarios, MCTM reduces the number of retrievals by 56.5% at most, and its query speed exceeds the baselines by up to 50.5% to 6.32 times.

Keywords: Trajectory Data Management, Mobile Edge Computing, Distributed search, Multi-Chord Distributed System

1 Introduction

Modern IoT and GPS technologies enable large-scale mobile trajectory data collection for applications like road network management, pathogen tracing, and route planning. However, managing such data requires addressing two key challenges: **reduce the latency and overhead of trajectory data transmission** and enabling **fine-grained representation and efficient querying**.

Reduce the latency and overhead of trajectory data transmission. Traditional cloud-centric approaches struggle with real-time demands, but Mobile Edge Computing (MEC) mitigates this by placing computation closer to data sources [11], significantly reducing latency and accelerating processing [15]. Unlike cloud computing, MEC leverages high-speed links between neighboring servers [14], minimizing multi-hop network delays while enhancing data accessibility and analytical capabilities.

Fine-grained representation and efficient querying. Many existing works lack fine-grained temporal recording. VRE [8] and TMan [6] use fixed intervals, leading to unnecessary trajectory access. TrajMesa [9] introduces the XZT index but may double data access. Spatial indexes also face efficiency challenges. R-tree [4] and its variants [1, 12] suffer from high latency on large data. XZ-Ordering [2] and $XZ2^+$ [10] mitigate this but introduce excessive empty query windows. Additionally, many spatio-temporal indexes have limitations. Dynamic tree-structured indexes, such as 3D-Rtree [13], incur excessive overhead due to frequent updates with large-scale data. Indexes inspired by XZ-Ordering, like $XZ2^+T$ [5], struggle to leverage the range query capabilities of underlying databases.

The contributions of this paper are summarized as follows:

– We propose a scalable multi-chord distributed system for efficient data transmission and querying. Trajectory data from mobile objects is stored on edge nodes, reducing network overhead and interconnected via skip lists.
– We introduce the TI index, the Index Cache, and the TSI index to improve the efficiency and accuracy of various types of queries, including temporal, spatial, ID-temporal, and spatio-temporal queries.
– We implemented and deployed the system, simulating the full data collection and query analysis process to validate its reliability. Comparative analysis with other baselines demonstrated the efficiency of our methods.

2 System Architecture

2.1 Multi-Chord Distributed System

In mobile scenarios, objects frequently switch between edge nodes, distributing trajectory data across multiple nodes. Since each edge node only records data within its coverage area, ID-temporal queries typically require broadcasting to all nodes, leading to inefficiency and high bandwidth overhead.

To address this, we propose using distributed skip list to track mobile objects' transmissions across edge nodes. Each edge node records the start and end times of a trajectory in a skip list node ($SkipNode$) and updates successor nodes as the object moves. This enables precise trajectory tracking for each object.

We organize nodes into a Chord DHT, storing the skip list's head node on the edge node corresponding to the hash of the object's ID. This simplifies updates and queries while supporting dynamic node addition for scalability. Each Chord DHT node acts as a virtual node, dynamically managing physical edge nodes to ensure load balancing, data backup, and fault tolerance. For ID-temporal queries, the system locates the head node of the skip list and forwards the request along the skip list to retrieve relevant trajectory data.

To handle long-distance communication for skip list updates, we introduce a multi-Chord system structure. Nodes in different regions form independent

Chord DHT subsystems, each maintaining its own skip lists. The head node of a skip list in each subsystem records the object's activity times across subsystems, enabling efficient cross-subsystem queries based on time ranges.

Fig. 1 illustrates this with regions R_1 and R_2 managed by subsystems S_1 and S_2. When object M enters R_1 at time t_0, S_1 creates skip list L_1 with head node H_1 at node B. When M moves to R_2 at time t_1, S_2 creates L_2 with head node H_2 at node X, recording H_1's location and M's activity times in R_1 (t_0, t_1). S_1 stops updating L_1, and node B records t_1 and H_2's location.

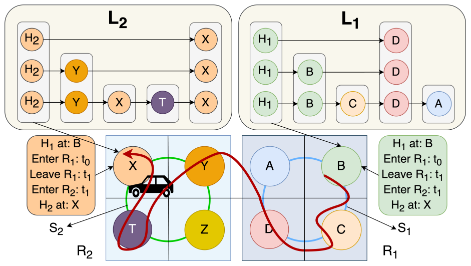


Fig. 1. Multi-Chord System Overview

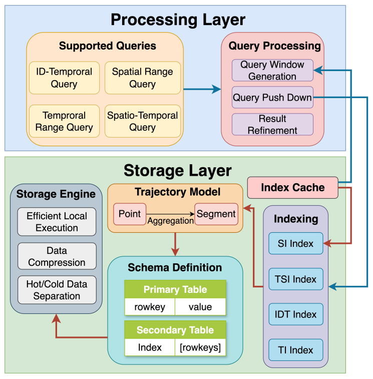


Fig. 2. Local System Framework

2.2 Local Framework

Fig. 2 illustrates the overall architecture of each edge node, which is divided into the processing layer and storage layer.

Processing Layer: Each edge node supports efficient query processing for various functions, including temporal range, spatial range, ID-temporal and spatio-temporal queries, detailed in Section 4.

Storage Layer: We use an efficient storage engine and compress trajectory data to minimize space usage. Outdated data is migrated to slower, more cost-effective storage. Secondary Tables store indexes, requiring just one data copy, further reducing storage costs. Both point-based and segment-based trajectory models are employed, with TI, SI, IDT, and TSI indexes supporting queries. Index Cache records SI index information, aiding in query window generation for spatial range and spatio-temporal queries, detailed in Section 3.

3 Storage Layer

3.1 Trajectory Model

The point-based model stores each point as a separate row, resulting in high storage costs and reconstruction overhead. The trajectory-based model stores the entire trajectory in one row, reducing storage but increasing false positives in spatial queries. Inspired by VRE [8], we use a segment-based model, dividing trajectories into continuous segments. This reduces storage, accelerates reconstruction, and improves query accuracy.

Since trajectory data points are periodically aggregated into segments, real-time queries and online analysis require a point-based model to temporarily store unaggregated points. Our work focuses on optimizing segment data storage and queries, so we adopt the point data management strategy from Geomesa [7] without further elaboration.

3.2 Temporal and ID-Temporal Indexing

Temporal Indexing Inspired by the TR index in TMan [6], we enhance the temporal index TI by incorporating segment type and start/end time descriptions for finer granularity. Therefore, TI has a structure of $\boldsymbol{Bin\ Num} + \boldsymbol{SegTime\ ID} + \boldsymbol{Segment\ Type} + \boldsymbol{Timepoint\ Num}$.

Bin Num: We partition the temporal dimension into non-overlapping **time bins** using a fixed **time period**: $BinNum(t) = \lfloor (t - InitialTime) \div BinLen \rfloor$

SegTime ID: Each time bin is further divided into **SegTimes** using a fixed **time duration**: $SegTimeID(t) = \lfloor ((t - InitialTime) \bmod BinLen) \div DurLen \rfloor$

A trajectory segment's interval is defined by its **Bin Num** and **SegTime**. Here, t is the segment's start time, $InitialTime$ is the timeline's start, and $BinLen$ and $DurLen$ are the time period and duration in seconds. Using one day ($BinLen = 86400$s) and one hour ($DurLen = 3600$s), a **2-byte Bin Num** can represent over 179 years, and **SegTime ID** requires only **5 bits**.

Segment Type: A full trajectory consists of multiple segments, categorized using 2 bits:

1. **End segment (type 0)**: the last segment when the trajectory terminates within the time interval. 2. **Mid segment (type 1)**: any segment that is neither the first nor last. 3. **Start segment (type 2)**: the first segment, including cases where the trajectory spans a short time within a single interval.

Timepoint Num: This records the start or end time of the trajectory segment using 9 bits, enhancing temporal filtering accuracy. For type 0, it records the end time; for type 2, the start time; and for type 1, no start or end time is recorded.

ID-Temporal Indexing We propose the IDT index to support ID-temporal queries. For a given segment, we first calculate its temporal index TI according to 3.2 and then combine TI with its *oid* (object id) to obtain the IDT index with a structure of $\boldsymbol{oid} + \boldsymbol{TI}$.

3.3 Spatial Indexing And Index Cache

TrajMesa [9] introduces the $XZ2^{+}$ index, dividing the area into $\beta \times \beta$ subregions and encoding them with a $\beta \times \beta$ bit PosCode. During spatial queries, it filters XElements across levels, combining valid PosCodes to form query windows.

However, many screened XElements lack trajectories (e.g., XE_1 in Fig. 3), leading to redundant PosCode checks and invalid query windows. Moreover, most PosCodes within an XElement do not contain trajectories. For instance, in XE_2, where only one subregion has a trajectory, $2^{\beta\times\beta-1} = 32768$ valid PosCodes must be checked among $2^{\beta\times\beta} = 65536$, causing unnecessary overhead.

To address this, we propose the **Index Cache**, which stores mappings between XElements and their PosCode lists as $(XElement, PosCode\ List)$ tuples. When generating an $XZ2^{+}$ index, it updates the PosCode List accordingly. To optimize cache space, LFU (Least Frequently Used) evicts rarely accessed tuples to disk. This improves query filtering, allowing $XZ2^{+}$ to serve as our spatial index SI efficiently.

3.4 Spatio-Temporal Indexing

Compared to the spatial dimension, the temporal dimension is naturally more continuous in searches. Therefore, we propose splitting the description of the temporal dimension into two parts of different granularity of representation, as introduced in 3.2: ***Bin Num*** and ***SegTime ID***.

We move the fine-grained description $SegTime\ ID$ to the end of the index, constructing the TSI index with a structure of ***Bin Num+SI+SegTime ID***. The coarse-grained $Bin\ Num$ at the front of TSI allows efficient pruning through the temporal dimension first when handling spatio-temporal queries. At the same time, the fine-grained $SegTime\ ID$ at the back enables range queries for cases where the query range covers multiple SegTimes during a single time bin. Compared to the spatial dimension, $SegTime$s within a single time bin are more continuous, generating many more range queries.

3.5 Storage Schema

The point-based trajectory model is a variant of the segment-based trajectory model, and our work mainly focuses on trajectory segments. Therefore, we only introduce the storage schema of the segment-based trajectory model, as shown in Fig. 4, which is divided into two parts:

Primary Table: This table uses the IDT index and stores trajectory segment data, including the object ID (*oid*), trajectory ID (*tid*), position points (*Points*), related indexes (TI, SI, TSI), and other segment attributes. The *Points* are serialized and compressed with zstandard [3] to reduce storage and transmission costs.

Secondary Table: Unlike TrajMesa [9] [10], where each index stores a copy of the trajectory data, we store the trajectory data only once in the Primary Table. The Secondary Tables store the relationships between indexes and row keys (IDT) in the Primary Table, reducing storage costs. When a new segment is inserted into the Primary Table, the corresponding indexes in the Secondary Tables are updated. During queries, the Secondary Tables filter out the IDTs, which are then used to retrieve the relevant segments from the Primary Table.

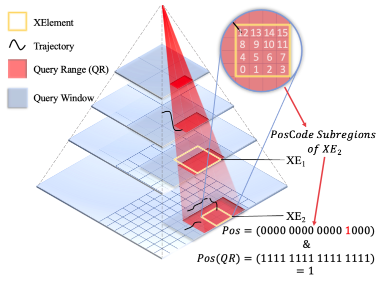
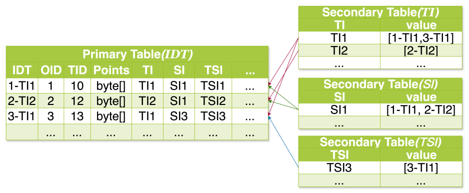


Fig. 3. $XZ2^+$ Query Process



Primary Table*(IDT)*

IDT	OID	TID	Points	TI	SI	TSI	...
1-TI1	1	10	byte[]	TI1	SI1	TSI1	...
2-TI2	2	12	byte[]	TI2	SI1	TSI2	...
3-TI1	3	13	byte[]	TI1	SI3	TSI3	...
	...	...	...	...	...	...	...

Secondary Table*(TI)*

TI	value
TI1	[1-TI1,3-TI1]
TI2	[2-TI2]
...	...

Secondary Table*(SI)*

SI	value
SI1	[1-TI1, 2-TI2]
...	...

Secondary Table*(TSI)*

TSI	value
TSI3	[3-TI1]
...	...

Fig. 4. Storage Schema

4 Query Processing

In MCTM, when receiving queries involving specific spatial ranges or target objects, each edge node determines its eligibility to execute and whether to forward the query. Each edge node manages a fixed spatial range and handles spatial range queries as follows: 1) forward the request to a neighbouring node closer to the given spatial range S_q, if S_q does not intersect with its range S_n; 2) execute the query locally and forward it to neighbouring nodes that meet the criteria, if S_q intersects with S_n; 3) execute solely, if S_q is contained within S_n. For ID-temporal queries, the edge node forwards the request to the appropriate nodes as outlined in Subsection 2.1.

Each edge node's local system handles queries through: 1) **Query Window Generation**: Generate query windows based on the request and indexing strategies in Section 3. 2) **Query Push Down**: Push the query windows down to the underlying database to trigger SCAN operations. 3) **Result Refinement**: Collect the results, remove unqualified segments and return the final results.

5 Experiments

5.1 Experiment Setting

We evaluated MCTM using a taxi trajectory dataset[3] (1.7 million records from 442 taxis, July 2013–June 2014). Spatial regions were partitioned into $1° \times 1°$ grids (Fig. 5), with high-density region A further divided into $0.5° \times 0.5°$ subregions, each managed by an edge node activated upon receiving data.

Edge nodes were simulated on three servers (8-core CPU, 32GB RAM, 512GB disk), while a separate server (4-core CPU, 16GB RAM) emulated taxis transmitting trajectory data chronologically. We adopted HBase as the underlying database for its scalability under high write loads, with coprocessors synchronizing indexes between Primary and Secondary Tables.

To assess query performance, 100 randomized spatial/temporal ranges per query type were generated, and results were averaged to minimize randomness and HBase cache effects.

[3] https://www.kaggle.com/datasets/crailtap/taxi-trajectory

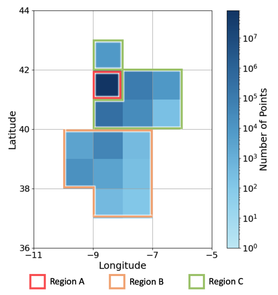


Fig. 5. Trajectory Distribution

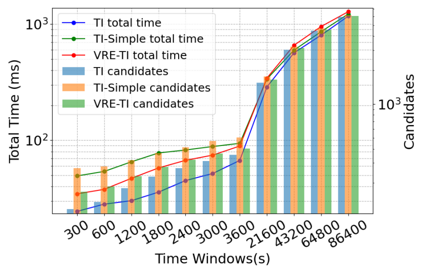


Fig. 6. Temporal Range Query Result

5.2 System Build Up

Trajectory messages activated edge nodes in regions A, B, and C, integrating them into the Chord DHT. As taxis moved across subregions, skip lists were dynamically updated, maintaining separate lists per region without data loss.

To handle high write loads, core region A activated backup nodes, forming clusters to enhance efficiency. Once the transmission was completed, the system stabilized, and the backup nodes were deactivated. We then issued corresponding query requests for the trajectory data in the subregions of the deactivated nodes and found that no data was lost. Instead, the deactivated nodes had automatically forwarded the data to other nodes for storage, ensuring that all queries were completed successfully.

5.3 Performance of Temporal Range Query

We compared our TI index with $VRE\text{-}TI$ [8] and the TR index's adjusted version tailored for segments. The TR index from TMan [6] uses fixed-length time bins, aligning with our time interval concept. We modified it to a $Bin\ Num + SegTime\ ID$ structure, termed $TI\text{-}Simple$. N $TI\text{-}Simple$ indexes match the precision of a TR index with N time bins, enabling a granularity comparison.

Fig. 6 shows that across time windows from 300 seconds to 86400 seconds, TI consistently retrieved fewer segment candidates by leveraging Segment Type and Timepoint Num for precise pruning. As the window increased, pruning efficiency slightly decreased, but TI remained the fastest and most accurate. Compared to $TI\text{-}Simple$, TI achieved **50.5%** higher speed and retrieved **56.5%** fewer candidates at 300 seconds.

5.4 Performance of ID-Temporal Query

We compared the IDT index with the ID-temporal index proposed by VRE [8], referred to as $VRE\text{-}IDT$, and a combination of object ID with $TI\text{-}Simple$ from Subsection 5.3, forming the structure $oid{+}TI\text{-}Simple$, named $IDT\text{-}Simple$ index. Fig. 7 presents the experimental results as the time window varies from

1800 s (30 min) to 86400 s (1 day). Our *IDT* index outperforms both in query speed and accuracy, attributed to its fine-grained temporal representation.

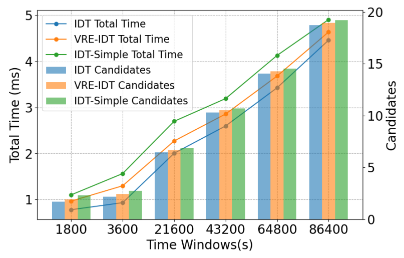


Fig. 7. ID-Temporal Query Result

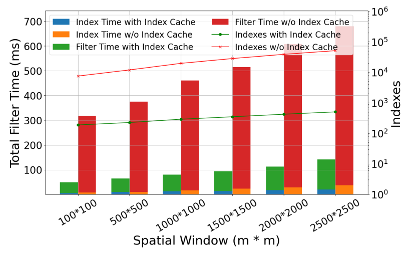


Fig. 8. Spatial Range Query Result

Table 1. Different PosCode Resolution Evaluation

β	2		3		4	
Index Cache	true	false	true	false	true	false
Index Time (ms)	**10.63**	10.73	**13.01**	16.48	**157.55**	1055.55
Filter Time (ms)	**28.56**	69.63	**67.71**	444.60	**151.70**	31262.12
Retrieve Time (ms)	22865.07	22754.37	11111.05	11130.36	10266.30	10211.78
Total Time (ms)	22904.26	22834.73	11191.77	11591.44	**10575.55**	42529.45
Indexes	**58**	549	**290**	19729	**734**	2731694
Candidates	83939		51316		**43036**	

5.5 Performance of Spatial Range Query

Different PosCode Resolution: With a fixed $1000m \times 1000m$ spatial window, we evaluated Index Cache across different PosCode resolutions β (Table 1). **Index Time** refers to the time required for index generation, **Filter Time** denotes the time spent on segment filtering in SITable, **Retrieve Time** represents the time taken to retrieve segments from the Primary Table, and **Total Time** corresponds to the overall query execution time.

The results demonstrate that Index Cache effectively accelerates spatial queries while mitigating access overhead. This advantage becomes more pronounced at higher PosCode resolutions, where it also enhances query accuracy.

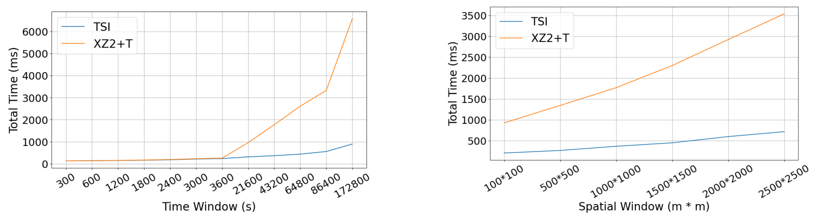
Different Spatial Windows: As shown in Fig. 8, when $\beta = 3$, **Index Time** and **Filter Time** increase with spatial window size due to more PosCode checks and index generation. Index Cache mitigates this, with optimization effects amplifying for larger windows. At $2500m \times 2500m$, speed improves by **3.8**$\times$, reducing indexes to **0.98%** of the non-cache case, with only **0.024%** additional storage overhead.

5.6 Performance of Spatio-Temporal Query

We compare our TSI index with $XZ2^{+}T$ from JUST-traj [5] over a one-hour period with $\beta = 3$, ensuring identical spatiotemporal granularity and candidate count. Thus, we focus on the improvement in query efficiency achieved by TSI.

Different Time Windows: With a fixed $1000m \times 1000m$ spatial window, we varied the time windows. As shown in Fig. 9(a), when the time window is within 3600 s, limited merging occurs in TSI. As it increases, TSI's range query optimization becomes more pronounced. At 172800 s (48 hours), TSI achieves a **6.32**$\times$ speedup over $XZ2^{+}T$.

Different Spatial Windows: With a fixed 43200 s time window, Fig. 9(b) shows query times across spatial windows. TSI consistently outperforms $XZ2^{+}T$ due to more range queries. At $2500m \times 2500m$, TSI is **4.92**$\times$ faster.



(a) Spatio-Temporal Query Result (Spatial Window is $1000m \times 1000m$)

(b) Spatio-Temporal Query Result (Time Window is 43200 s)

Fig. 9. Spatial Range Query Result

6 Conclusion

This paper presents MCTM, a multi-chord distributed system for efficient trajectory data transmission and queries in mobile edge computing. To our knowledge, it is the first large-scale trajectory management system designed for this context. We introduce a fine-grained temporal index TI, a spatio-temporal index TSI leveraging range queries from underlying databases, and an Index Cache that reduces query windows for a given spatial range. Future work will focus on optimizing MCTM and expanding its support for additional trajectory query tasks.

References

1. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The r*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data. pp. 322–331 (1990)
2. BÖxhm, C., Klump, G., Kriegel, H.P.: Xz-ordering: A space-filling curve for objects with spatial extension. In: International Symposium on Spatial Databases. pp. 75–90. Springer (1999)
3. Collet, Y., Kucherawy, M.: Zstandard compression and the application/zstd media type. Tech. rep. (2018)
4. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on Management of data. pp. 47–57 (1984)
5. He, H., Li, R., Bao, J., Li, T., Zheng, Y.: Just-traj: A distributed and holistic trajectory data management system. In: Proceedings of the 29th International Conference on Advances in Geographic Information Systems. pp. 403–406 (2021)
6. He, H., Xu, Z., Li, R., Bao, J., Li, T., Zheng, Y.: Tman: A high-performance trajectory data management system based on key-value stores. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE). pp. 4951–4964. IEEE (2024)
7. Hughes, J.N., Annex, A., Eichelberger, C.N., Fox, A., Hulbert, A., Ronquest, M.: Geomesa: a distributed architecture for spatio-temporal fusion. In: Geospatial informatics, fusion, and motion video analytics V. vol. 9473, pp. 128–140. SPIE (2015)
8. Lan, H., Xie, J., Bao, Z., Li, F., Tian, W., Wang, F., Wang, S., Zhang, A.: Vre: a versatile, robust, and economical trajectory data system. Proceedings of the VLDB Endowment **15**(12), 3398–3410 (2022)
9. Li, R., He, H., Wang, R., Ruan, S., He, T., Bao, J., Zhang, J., Hong, L., Zheng, Y.: Trajmesa: A distributed nosql-based trajectory data management system. IEEE Transactions on Knowledge and Data Engineering **35**(1), 1013–1027 (2021)
10. Li, R., He, H., Wang, R., Ruan, S., Sui, Y., Bao, J., Zheng, Y.: Trajmesa: A distributed nosql storage engine for big trajectory data. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE). pp. 2002–2005. IEEE (2020)
11. Siriwardhana, Y., Porambage, P., Liyanage, M., Ylianttila, M.: A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects. IEEE Communications Surveys & Tutorials **23**(2), 1160–1192 (2021)
12. Tao, Y., Papadias, D., Sun, J.: The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In: Proceedings 2003 VLDB conference. pp. 790–801. Elsevier (2003)
13. Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-temporal indexing for large multimedia applications. In: Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems. pp. 441–448. IEEE (1996)
14. Xia, X., Chen, F., He, Q., Grundy, J., Abdelrazek, M., Jin, H.: Online collaborative data caching in edge computing. IEEE Transactions on Parallel and Distributed Systems **32**(2), 281–294 (2020)
15. Zhou, F., Chen, H.: Decs: Collaborative edge-edge data storage service for edge computing. In: Collaborative Computing: Networking, Applications and Worksharing: 16th EAI International Conference, CollaborateCom 2020, Shanghai, China, October 16–18, 2020, Proceedings, Part I 16. pp. 373–391. Springer (2021)