BT-Tree: A Reinforcement Learning Based Index for Big Trajectory Data

TU GU, Nanyang Technological University, Singapore
KAIYU FENG, Beijing Institute of Technology, China
JINGYI YANG, Nanyang Technological University, Singapore
GAO CONG, Nanyang Technological University, Singapore
CHENG LONG, Nanyang Technological University, Singapore
RUI ZHANG, Huazhong University of Science and Technology, China

With the increasing availability of trajectory data, it is important to have good indexes to facilitate query processing. In this work, we propose BT-Tree, which is built through a recursive bi-partitioning approach, for the processing of range and KNN queries for past trajectory data. We first propose a cost function based method (CFBM) to build the BT-Tree. Specifically, we design a novel cost function, which incorporates the characteristics of both the data and historical query workload, to decide how to partition a BT-Tree node. Then we propose a reinforcement learning (RL) based method to address CFBM's limitations, such as making locally optimal decisions that may lead to global suboptimality. Experiments on three real datasets with up to 800 million data points show that the CFBM generally outperforms the baselines in terms of query processing time and the RL based method consistently outperforms the baselines and has more significant advantages on larger datasets.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Spatio-temporal data, spatio-temporal query processing, learned index, deep learning, reinforcement learning

ACM Reference Format:
Tu Gu, Kaiyu Feng, Jingyi Yang, Gao Cong, Cheng Long, and Rui Zhang. 2024. BT-Tree: A Reinforcement Learning Based Index for Big Trajectory Data. *Proc. ACM Manag. Data* 2, 4 (SIGMOD), Article 194 (September 2024), 27 pages. https://doi.org/10.1145/3677130

1 Introduction

The use of GPS devices has been increasingly popular for the past years resulting in the collection of an increasing amount of trajectory data. A trajectory contains a sequence of 3-dimensional data points, each of which consists of a timestamp and the longitude and the latitude of a location. Two adjacent data points form one segment. To support efficient query processing on trajectories, such as range queries (e.g., the total number of birds that fly into a particular forest in summer) and KNN queries (e.g., the locations of the 100 nearest taxis to a particular taxi stand during peak hour), many indexes for trajectory data [4, 6, 26, 38, 41, 48] have been designed. However, it is still an open issue

Authors' Contact Information: Tu Gu, gutu0001@e.ntu.edu.sg, Nanyang Technological University, Singapore; Kaiyu Feng, kaiyufeng@outlook.com, Beijing Institute of Technology, China; Jingyi Yang, jingyi006@e.ntu.edu.sg, Nanyang Technological University, Singapore; Gao Cong, gaocong@ntu.edu.sg, Nanyang Technological University, Singapore; Cheng Long, c.long@ntu.edu.sg, Nanyang Technological University, Singapore; Rui Zhang, rayteam@yeah.net, Huazhong University of Science and Technology, China.

ACM 2836-6573/2024/9-ART194
https://doi.org/10.1145/3677130

Table 1. Characteristics of Different Trajectory Data indexes

Index Name	TrajStore [6]	GCOTraj [48]	SETI [4]	SEB & CSB-Tree [41]	TB & STR-Tree [26]	3DR-Tree [38]	BT-Tree
Partition along any dimension at any location based on data features?	No	No	No	No	No	No	Yes
Consider creation of new trajectory segments?	Yes	No	No	No	NA	NA	Yes
Leverage on historical workload?	Yes	Yes	No	No	No	No	Yes

how to index trajectories due to its complexity with a sequence of points with both spatial and temporal values. The state-of-the-art methods typically divide trajectories into trajectory segments and index the segments. The challenge is how to segment trajectories and how to organize the segments for better query performance.

Existing indexes for past trajectory data could be divided into two categories, i.e., the data partitioning based and the space partitioning based indexes. Data partitioning based indexes [26, 36, 38] work in a similar way as R-Tree [12] and enclose trajectory segments using 3-dimensional Minimum Bounding Boxes (MBBs). The space covered by the MBBs usually has excessive overlaps among trajectory segments which then causes the indexes to be unable to effectively prune irrelevant data. Space partitioning based indexes [4, 6, 41] usually deal with the spatial dimensions and the temporal dimension of trajectory data separately. Indexes in this category usually consist of a spatial index at the top followed by a temporal index at the bottom. On the other hand, there also exists GCOTraj [48] that partitions the trajectory data into grid cells and then uses space filling curves [10] based ordering to store them on disk pages. Although space partitioning based trajectory indexes generally have better query performance than data partitioning based trajectory indexes, they still have various limitations. *L1*: The performances of the indexes [4, 6, 41] that consist of a spatial index at the top and a temporal index at the bottom tend to deteriorate significantly when processing range queries with more selective temporal predicates. In an extreme scenario, if a query has selective temporal predicate and no spatial predicate, the spatial index will not be able to prune any irrelevant data. *L2*: Space partitioning unavoidably cuts some trajectory segments resulting in the formation of new segments, which has a large impact on the query performance of the index as explained in [6]. However, previous work [4, 41, 48] usually does not factor in the creation of new segments when deciding how to partition the data space. *L3*: Among proposals [6, 48] that leverage on historical query workload to build the indexes: TrajStore [6] only takes into consideration the average query size while in fact queries may have different sizes and are not necessarily uniformly distributed; the grid size of GCOTraj [48] is fixed and independent of the query workload leading to poor query performance when the queries are not uniformly distributed. We summarize the characteristics of different indexes for past trajectory data in Table 1.

Index Design. We propose BT-Tree (Section 3), which is a binary tree that is constructed through a recursive bi-partitioning approach. In a BT-Tree, each node corresponds to some sub-space of the 3-dimensional trajectory data space and the root of the tree corresponds to the entire data space. In addition, each internal tree node can be partitioned along any dimension at any location (solution to limitation *L1*). To the best of our knowledge, our work is the first that allows different parts of the trajectory data space to be flexibly partitioned along any dimension at any location, based on the characteristics of the data and historical query workload. Previous work (e.g., [4, 6, 41]) mostly adopts a rigid spatial-first approach (i.e., to partition the nodes at the upper layers along the spatial dimensions *only*). In Figure 1, we illustrate how our proposed BT-Tree processes range query q, by pruning irrelevant nodes starting from the root N_R.

Index Construction. A key challenge that has to be overcome to build a BT-Tree is *how to select the best cut to partition a tree node?* It is crucial to use a quantifiable measure to identify the best cut to partition a tree node. One quantifiable measure used in previous work is query skew introduced

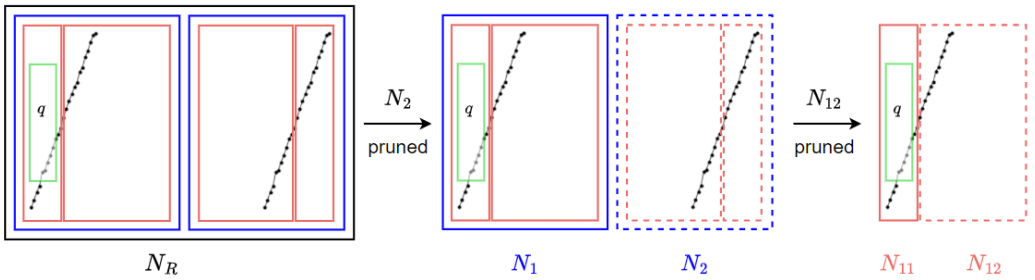


Fig. 1. Illustration of How BT-Tree Processes Range Query

in [8], which experimentally shows that query skew reduction leads to better query performance for multidimensional indexes. The intuition is that if two adjacent regions in the data space contain queries that are selective over different dimensions, the two regions as a whole are considered to have a large query skew and it is ideal to separate them with a cut so that the query skew for each partition is reduced significantly. However, our experiments (Section 6.2.7) show that the direct application of this idea on trajectories does not work because it does not take into consideration important characteristics of trajectory data. Therefore, we propose the concept of data-aware query skew, which takes data distribution into consideration when measuring query skew. We then propose a greedy approach (Section 4), which we call the cost function based method (CFBM), to build BT-Tree using a hand-crafted cost function that considers the characteristics of both the trajectory segments (solution to limitation *L2*) and historical query workload (solution to limitation *L3*) when choosing the best cut. Our designed cost function reflects the increase in the number of trajectory segments and the impact on data-aware query skew caused by a cut.

One major limitation of the CFBM is that it makes locally optimal decisions at internal nodes without fully exploiting the global impact of these decisions on the tree quality. To address this limitation, we propose a reinforcement learning (RL) based method (Section 5) to train an RL model to build BT-Tree. We train the RL model by repeatedly constructing trees and gradually learning to identify good cuts through rewards. *To make the RL based method work, we have to overcome two main challenges, including the RL model design and the large overhead incurred by model training.* First, in our design of the action space, we use some heuristic rule to exclude "bad" actions from the action space, which makes exploration during model training more effective and efficient. In addition, we design a reward function to train the RL agent to take "good" actions by comparing the current RL policy with an existing method. Second, due to the large overhead of training the RL model on large datasets, we want to carefully design the state space (for RL agent's decision making) in a way such that we are able to train the RL model on a small subset of a large dataset and then to apply it on the large dataset. A possible idea is to use the extreme values of the tree node MBB as the state metrics, as used in [49]. However, our preliminary experiments showed that this state design did not work well. One possible reason is that this information does not capture any useful feature of the data and hence is insufficient to be generalized to larger datasets. Note that similar MBBs may contain significantly different data (such as the number of trajectories) on the small dataset and the large dataset. As a result, the optimal cut for the same MBB may be different on the small dataset and the large dataset. In our design, the state contains two important pieces of information. (1) The impact of each candidate cut, including the increase in the number of trajectory segments and change in data-aware query skew caused by each action. (2) A spatial embedding vector of the data, which captures the unique spatial characteristics of the data. As the state metrics in (1) are ratios (percentage values) in the range of $[0, 1]$, they are not expected to differ significantly on the small training dataset and the large dataset, since the data distribution

remains mostly the same. In addition, the presence of spatial embedding enables the RL model to learn the spatial features of the data and then to learn how a cut in a specific region eventually affects the tree quality. This explains why an RL model trained on a small dataset can be used on a large dataset.

Contributions. In summary, we make the following contributions:

(1) We propose a new index, BT-Tree, for past trajectory data. To the best of our knowledge, this is the first work that allows different parts of the trajectory data space to be flexibly partitioned along any dimension at any location, based on the characteristics of the data and historical query workload, instead of adopting a rigid spatial-first approach (Section 3). We develop a new cost function, which enables us to select good cuts to partition BT-Tree nodes, by measuring their impacts on the trajectory data and the query workload. This cost function based method (CFBM) builds a BT-Tree efficiently, and achieves good query performance (Section 4). one additional benefit of the BT-tree is that the currently deployed query processing algorithms are still applicable and the proposed index can be easily deployed by current databases.

(2) To address the limitations of the CFBM, we propose an RL based method to build the index. The technical contribution lies in the new design of the action space, the state space and the reward functions. Our design enables the trained RL model to select good cuts. Moreover, it enables the RL model to be trained effectively and efficiently on a small dataset and then applied on a large dataset. To the best of our knowledge, this is the first work that uses machine learning to build an index for past trajectory data for the processing of range and KNN queries. (Section 5)

(3) Extensive experiments on three real datasets show: (a) Our RL based method achieves up to 93% better query performance for range queries and 84% for KNN queries than the baselines. (b) A BT-Tree constructed by an RL model that is trained using one workload can readily process queries with different sizes and distributions. Despite the change of workload, it is still able to consistently outperform all baselines and its query performance drops only slightly compared with one that is trained and tested on the same query workload. (c) Our RL based method scales well with dataset size and its advantage becomes more significant on larger datasets. (Section 6)

2 Related Work and Preliminary

2.1 Trajectory Data Indexing

Data Partitioning Based Indexes. 3DR-Trees [36, 38] use three-dimensional MBBs to enclose trajectory segments in a trajectory dataset. STR-Tree and TB-Tree [26] are considered 3DR-Tree variants. The TB-Tree ensures that a leaf node only contains segments from the same trajectory while the STR-Tree uses a different strategy to produce the MBBs. The TB-Tree generally outperforms STR-Tree when processing range queries. X-FIST [31] is another recently proposed trajectory data index in this category of indexes. However, X-FIST is designed to handle similarity search (which is to find trajectories from a database, which are similar to a query trajectory) and is unable to return accurate query results for range queries or KNN queries. Therefore, it is not compared to as a baseline in our experiments. Classic methods for partitioning spatial data along different dimensions such as KD-Tree [3] and P^+-tree [54] have been used on many types of data including time series data [55] and moving objects for the prediction of their future locations [1]. However, to the best of our knowledge, they have not been used for past trajectory data, for the processing of range queries, which is the query type we focus on in this work.

Space Partitioning Based indexes. Space partitioning based indexing methods for past trajectory data usually handle the spatial dimensions and the temporal dimension of a trajectory dataset separately. These methods are more commonly used and generally demonstrate better query performances than data partitioning based indexing methods. SETI [4] first uses a grid to divide the spatial data space into static and non-overlapping cells. Then in each cell, the trajectory segments

contained are indexed by the temporal dimension using a 1-dimensional R*-Tree [2]. The SEB-Tree and the CSE-Tree, both proposed in [41], are also constructed using a similar approach which indexes the spatial dimensions at the top followed by the temporal dimension at the bottom. MGeohash [20] indexes spatiotemporal data by first partitioning the data into time pieces and then building a grid based spatial index on each time piece. Note that MGeohash is designed for point data and usually returns different (and inaccurate) query results compared to most existing trajectory data indexes (e.g., TrajStore and TB-Tree). Therefore, MGeohash is not compared to as a baseline in our experiments. TrajStore [6] is arguably the most commonly used space partitioning based trajectory data index built based on historical query workload. The high level idea is to first build a spatial index using a quadtree. Each cell in the quadtree corresponds to a collection of one or more disk blocks that contain the trajectory segments in the cell. Trajectory segments in the same cell are grouped together and stored in disk blocks based on their end-time-stamps. There also exist other space partitioning based indexes designed for other tasks, such as trajectory compression [44], and trajectory joins [35]. They are beyond the scope of this paper.

Limitations of Existing Methods. Data partitioning based trajectory indexes are not commonly used as they are unable to effectively prune irrelevant data because of excessive overlaps between the MBBs. The three limitations of space partitioning based indexes have been discussed in Section 1.

Other Trajectory Data Management Work. There also exists work [27, 33, 43, 51] that proposes indexes for road network based trajectory data. However, these indexes are reliant on the availability of road networks and the performance of map matching techniques and are unable to handle free space based trajectory data. There are several recent systems [15, 19, 34, 39, 45, 46, 52] developed for various trajectory data management tasks. SQUID [52] proposes to build temporal partitions at the top followed by a spatial index (e.g., an R-Tree [12]) for each temporal partition at the bottom. Although SQUID is not designed for the query types considered in this work, we show how SQUID's temporal first indexing approach performs when processing range queries in our experiments. The other work [15, 19, 34, 45, 46] does not propose any index that is suitable to be used to handle the query types considered in this paper. For example, Tinba [39] is designed to incrementally maintain partitions under the updates of trajectory data for the processing of trajectory similarity queries, such as DTW [42]. It focuses on the selection of partitions that need to be reorganised from the set of candidate partitions under updates, using only existing partitioning methods such as STR [16] without splitting the trajectories. It was shown in previous work [26, 36, 38] that the methods using whole trajectories for partitioning perform worse than TB-tree for range and KNN queries, which is a baseline in our experiments.

Indexes for Moving Objects. Indexes for moving objects (e.g., [5, 24, 25, 32, 37, 50, 53]) appear related to indexes for past trajectory data. However, they are fundamentally different as indexes for moving objects focus on predictions of future locations of moving objects based on certain assumptions, and often consider velocities of individual objects when indexing them. For example, TPR-Tree [32] and TPR*-Tree [37] index moving objects based on the R-Tree structure by taking into consideration both their current locations and their velocities. These indexes are not suitable for indexing past trajectory data, and previous work that indexes past trajectory data does not compare to these indexes for moving objects.

2.2 Learning Based Spatial Data Management

Learned indexes usually learn a function that maps a search key to the storage address of a data object. The idea is first introduced in [14], which proposes the Recursive Model Index (RMI) to learn a cumulative distribution function (CDF) using a neural network to predict the rank of a search key. Subsequently, a number of works [7, 8, 17, 18, 23, 28, 40, 47] have proposed novel methods to index

spatial or multidimensional data. The Z-order model [40] extends RMI to spatial data by using a space filling curve to order data points and then learning the CDF. Recursive spatial model index (RSMI) [28] further develops the Z-order idea [40] and RMI. LISA [18] is a disk-based learned index that partitions the data space with a grid, numbers the grid cells, and learns a data distribution based on this numbering. Flood [23] also maps a dataset to a uniform rank space before learning a CDF. Differently, it utilizes workload to optimize the learning of the CDF, and it learns the CDF of each dimension separately. Tsunami [8] overcomes the limitation of Flood in handling skewed workload. Note that these learned indexes are designed for point data. They cannot be used to query trajectory data, where each trajectory is a sequence of points, rather than one multidimensional point. One possible way to use existing CDF based indexing methods to index trajectory data could be to directly index the data points. However, this is not a common practice to handle trajectory data and leads to inaccurate results when processing trajectory queries, such as range queries, which is a query type considered in our paper. In contrast, our proposed BT-Tree is designed for past trajectory data. We model the recursive bi-partitioning process as a Markov Decision Process (MDP) and carefully design the action space, the state space and the reward signal.

2.3 Preliminary

Tsunami [8] proposes a method to measure query distribution, which is known as query skew. It experimentally shows that partitioning data in a way that maximizes query skew reduction improves query performance. For example, given temporal data space $[0, 1]$, if 50% of the queries have selective temporal predicates within time interval $[t', 1]$ while the remaining 50% do not have any selective temporal predicate, it is intuitive to partition the data along the temporal dimension at $t = t'$. The two resulting partitions will have smaller query skew and can be further partitioned differently based on the features of the queries in their respective regions. We first briefly explain how query distribution is handled in [8], and then discuss about how we are motivated by the idea to build BT-Tree in later sections. The skew of a query set Q with respect to a range $[a, b]$ along dimension i is given by

$$Skew_i(Q, a, b) = Diff(UNI_i(a, b), PDF_i(Q, a, b)) \tag{1}$$

where $UNI_i(a, b)$ is a uniform distribution over $[a, b]$, $PDF_i(Q, a, b)$ is the Probability Density Function of queries in Q over $[a, b]$ and $Diff$ measures the difference between the two distributions. First, $PDF_i(Q, a, b)$ is approximated using a histogram (known as $Hist$) as follows. Range $[a, b]$ is first divided into n subranges of the same size. The subrange number of e ($a \leq e \leq b$) is represented by $SRN(e)$, which means e falls into the $SRN(e)^{th}$ subrange. If a query $q \in Q$ intersects with m consecutive subranges, then it contributes $1/m$ unit of mass to each subrange. This approximation of $PDF_i(Q, a, b)$ is $Hist_i(Q, a, b, n)$, which is a vector of length $(SRN(b) - SRN(a) + 1)$. Then, $UNI_i(Q, a, b)$ is computed. It is also a vector of length $(SRN(b) - SRN(a) + 1)$, each entry in which is the average value of the entries in $Hist_i(Q, a, b, n)$. Finally, the query skew is computed following equation 1.

As explained in [8], query skew has to be computed for different query types separately. This is because the skew of different query types can cancel each other out leading to inaccurate computations of query skew. Therefore, the query workload is divided into different query types based on the number of subranges each query covers along each dimension. Each query belongs to exactly one query type. Query skew is then computed by summing up the query skew for each query type.

Remark. Tsunami's approach to compute query skew has two major limitations. Firstly, it is based on query distribution alone and is independent of the data. This approach may lead to meaningless computation of query skew, i.e., when a query intersects with a few subranges but does not intersect

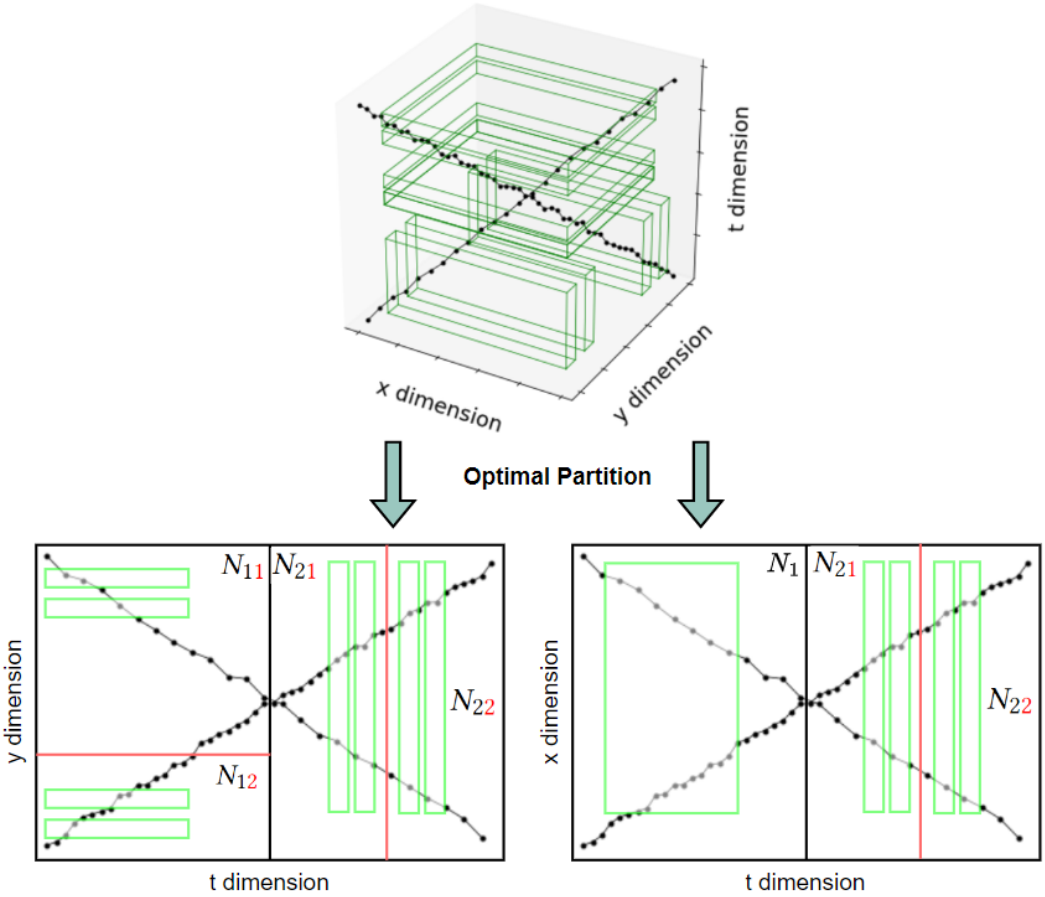


Fig. 2. Motivation for Indexing Different Parts Differently

with any data. Secondly, Tsunami's approach to group queries does not distinguish between queries that have similar sizes but cover different regions of the data space. This approach usually leads to inaccurate computation of query skew, as the query skew of queries located in different regions of the data space can cancel each other out. Detailed discussions are covered in Section 4.3.2. In contrast, we propose data-aware query skew, which is a different approach to measure query skew by considering both the data and the query workload. We also propose a new method to better reflect the difference among queries. Details will be presented in Section 4.3.

3 BT-Tree

Problem Definition. Given a dataset D, containing past trajectory data, we aim to build a disk based index, to minimize the I/O cost incurred for query processing. Note that we follow the setting of previous work (e.g., [6, 48]) and propose an index for past trajectory data that focus on movement in free space, based on historical query workload Q.

As explained in Section 2.1, a spatial (resp. temporal) first indexing approach leads to poor query performance when the queries have highly selective temporal (resp. spatial) predicates. Therefore, we propose a new structure, BT-Tree, that allows different regions in the data space to be flexibly indexed along any dimension at any location based on characteristics of the data and the query

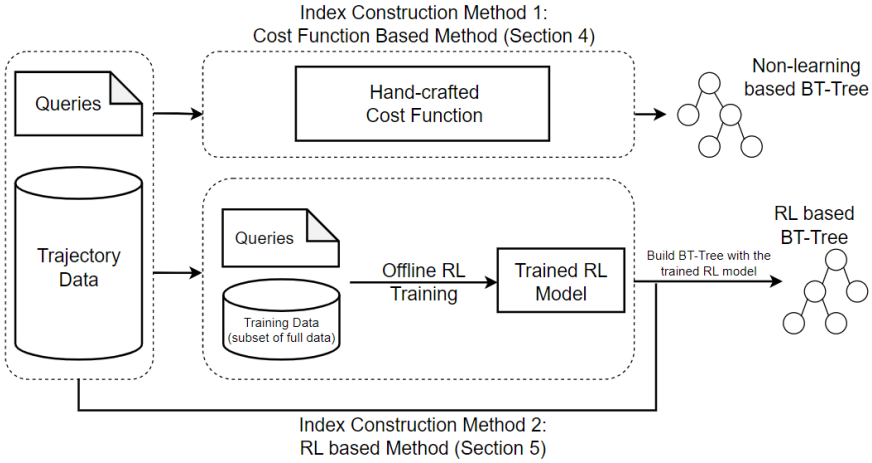


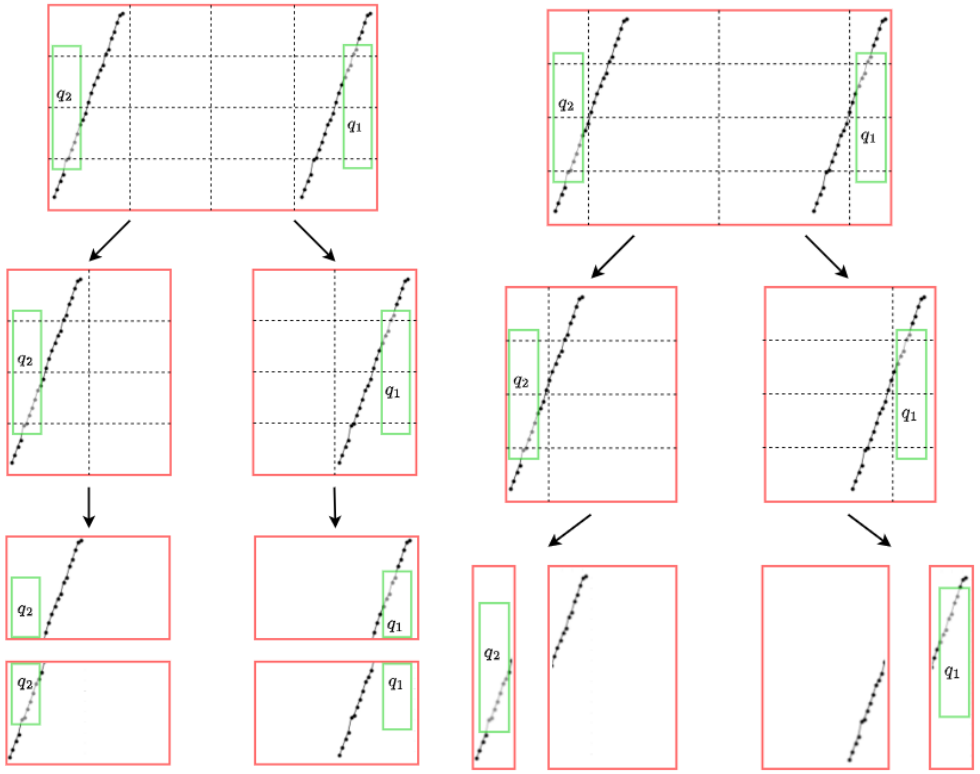
Fig. 3. BT-Tree Overview

workload, instead of adopting a rigid spatial-first indexing approach as it is in most of existing work.

We use Figure 2 to illustrate how our proposed approach is better than a rigid spatial-first approach. When deciding how to recursively bi-partition the trajectory data space, the optimal cuts at different internal nodes vary and depend on the query workload (green cubes). For example, when cutting nodes N_1 and N_2 (resulting child nodes of the black cut), due to different characteristics of the queries in these regions, i.e., queries in N_1 are more selective over dimension y while queries in N_2 are more selective over dimension t, they are cut along dimensions y and t respectively. The resulting tree is optimal as each query intersects with a minimum number of leaf nodes. For clear illustration, we show the tree and the queries through projections onto the $x-t$ and $y-t$ planes respectively. *It shows that to build an optimal tree, different parts of the data space should be flexibly indexed along any dimension at any location depending on the data and query features.*

A BT-Tree is a trajectory data index constructed by a recursive bi-partitioning approach. Each node corresponds to a subspace of the 3-dimensional trajectory data space and is represented as a minimum bounding box (MBB) of all its child nodes. An MBB is represented by the two extreme values of each dimension, i.e., $(x_{min}, x_{max}, y_{min}, y_{max}, t_{min}, t_{max})$. Each node holds all the trajectory segments that are fully contained by the MBB. The root node represents the whole data space. If an original trajectory segment intersects more than one MBB, it will be cut into smaller segments such that each smaller segment is covered by only one MBB. Each internal node has two child nodes which are obtained by bi-partitioning the parent node along a selected dimension at a selected value. Each leaf node represents a block that contains up to M trajectory segments and is stored on a disk page.

Note that the construction of BT-Tree leverages on the characteristics of the data and historical query workload when deciding how to cut a tree node. The problem to solve is how to choose the dimension and the location to partition every BT-Tree internal node. As shown in Figure 3, given a trajectory dataset and a query workload, we propose two different ways to construct BT-Tree. Specifically, in Section 4, we propose a greedy method which relies on a hand crafted cost function. In Section 5, we propose an RL based method.



(a) Tsunami's Approach (b) Our Proposed Approach

Fig. 4. Data Partitioning and Query Skew Reduction

4 BT-Tree Construction (Greedy)

4.1 Motivation

It is ideal to build a BT-Tree that minimizes the I/O cost when processing the queries in a given workload. However, this is not possible in practice because the I/O cost incurred to process a query can be measured only after the index is built. As a result, at an internal node, it is not possible to know which cut will ultimately lead to the minimization of I/O cost in the final index. An intuitive greedy approach, which we call **MinNode**, is to always pick the cut that minimizes the number of queries that intersect with both resulting child nodes. Although this method achieves some improvement in query performance (as shown in Section 6.2.2), it incurs large overhead in index construction (as shown in Section 6.2.9), because at every internal node, we need to use every query in the workload in order to determine the locally optimal cut.

To enable BT-tree to choose different good dimensions to partition different parts of the data space, we propose a new way to compute query skew, which we call data-aware query skew, based on the characteristics of both the data and the query workload, and use the data-aware query skew to guide the selection of partitioning dimensions for different parts of data space. The high level idea is to create subranges (which also determine the candidate cuts for partitioning) each of which contains a similar number of data points so as to build a more balanced tree to compute data-aware query skew. As shown in Figure 4, Tsunami [8] creates highly unbalanced subranges (boundaries are the dotted lines), some of which even contain no data, since it does not consider data distribution. We observe that our proposed approach (Figure 4b) achieves better query performance as queries

q_1 and q_2 intersect with a smaller number of leaf nodes. An experiment to compare our proposed data-aware query skew and Tsunmai's query skew can be found in Section 6.2.7.

Additionally, relying on query skew reduction alone is insufficient to build a good BT-Tree. This is because when applying space partitioning on trajectory data, certain trajectory segments will be cut resulting in the creation of new segments (limitation *L2* in Section 1). As explained in [6], the increase in the number of trajectory segments resulted from a partitioning process has a large impact on the query performance of the index. However, this impact is not captured in the computation of query skew. As a result, we would like to design a new cost function that reflects (1) data-aware query skew reduction, and (2) creation of new trajectory segments resulted from a cut. In the remaining of the section, we first discuss the design of the cost function (Section 4.2), and then explain how to use it to build BT-Tree (Section 4.3). We call this BT-Tree construction method the cost function based method (**CFBM**).

4.2 Cost Function

We propose to use the following function, CF, to measure the cost of cut cut at a BT-Tree node along dimension i:

$$CF_i(cut) = (1 - \lambda) \cdot (1 - QSred_i(cut)) + \lambda \cdot SegInc_i(cut) \tag{2}$$

where $QSred(cut)$ is the percentage data-aware query skew reduction achieved by cut, $SegInc(cut)$ is the percentage increase in the number of trajectory segments caused by cut and λ is a pre-defined weight parameter. Note that details about data-aware query skew will be shared in Section 4.3. We will test different values of λ and evaluate its impact on the index in Section 6.2.1. Intuitively, among candidate cuts at a BT-Tree node, a cut with a smaller value of CF is considered better than a cut with a larger value of CF.

4.3 BT-Tree Construction

Given a trajectory dataset D and a query workload Q, we aim to build a BT-Tree using our proposed cost function CF. When building a BT-Tree, we aim to minimize CF when deciding how to partition tree nodes. A summary of the index construction process is as follows. (1) Construct the set of n subranges $SubR_i$ for each dimension $i \in \{x, y, t\}$ (Section 4.3.1). (2) Divide the given queries into different query types. This is to ensure that the skew of different query types does not cancel each other out so as to avoid inaccurate computations of query skew, as illustrated in Section 4.3.2. Note that steps (1) and (2) incorporate data features as elaborated in Sections 4.3.1 and 4.3.2. We call the resulting query skew data-aware query skew. (3) Recursively bi-partition the data space until no BT-Tree node contains more than M trajectory segments (Section 4.3.3).

4.3.1 Subrange Construction. We construct the set of n subranges $SubR_i$ for a trajectory dataset D for each dimension $i \in \{x, y, t\}$ independently. The process is as follows. (1) Sort all data points in ascending order using their values of dimension i. (2) Obtain the n subranges and their corresponding boundaries such that each of the first $n - 1$ subranges contain $\lfloor \frac{|D|}{n} \rfloor$ data points and the last subrange contains the remaining data points. Our approach ensures that each subrange contains a similar number of data entries so as to build a more balanced BT-Tree.

4.3.2 Query Clustering. We divide the query workload into different query types through the following steps. (1) Each query q is represented by a vector of length $3n$, F_q, which is a concatenation of mass contributions of q along each dimension $i \in \{x, y, t\}$. If there are a total of 4 subranges along dimension i and q intersect with the 2 subranges in the middle, its mass contribution along i will be $(0, \frac{1}{2}, \frac{1}{2}, 0)$. (2) DBSCAN [9] is then used to group the queries using their corresponding F_q. This clustering step enables us to group queries based on their query dimensions, sizes and locations in the data space, all of which are reflected in F_q. Note that the choice of clustering algorithms does not have a significant impact on the computation of data-aware query skew. We then compute

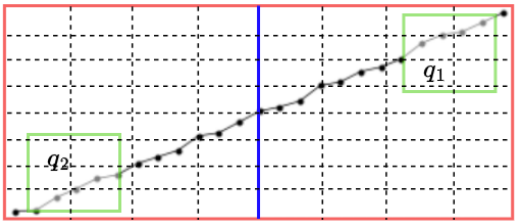


Fig. 5. Illustration of Query Skew Reduction

the vector $Hist_i(Q, a, b, n)$ where a and b are the lower and upper bounds of the data space along dimension i so that data-aware query skew reduction and CF for a cut at a tree node in the BT-Tree building process can be obtained in $O(1)$ time.

We use the following example (Figure 5) as an illustration of query clustering. We use the horizontal dimension as an example. The data space is divided into 8 subranges along the horizontal dimension and the query workload Q consists of q_1 and q_2. Following Tsunami's method, they belong to the same query type because they have the same size and query over the same dimensions, although they are located far apart. Following our method, they have different mass contributions ($q_1 : (\frac{1}{2}, \frac{1}{2}, 0, 0, 0, 0, 0, 0)$ and $q_2 : (0, 0, 0, 0, 0, 0, \frac{1}{2}, \frac{1}{2})$) along the horizontal dimension and hence belong to different query types. Based on data-aware query skew, the blue cut is selected, which is a good cut to partition the node as each query intersects with only one resulting child node. However, Tsunami's method obtains negative query skew reduction for the blue cut, which makes no sense. This is because Tsunami's method fails to group the queries properly. It classifies q_1 and q_2 as the same query type which causes their skews to cancel each other out.

4.3.3 Index Construction. A BT-Tree is constructed through a recursive bi-partitioning approach. As explained earlier, we flexibly index different parts of the data space along any dimension at any location based on the features of the query workload and the data. Specifically, at a node that contains more than M trajectory segments, we choose the cut that minimizes CF to partition it. $QSred$ (in equation 2) for a cut cut along dimension i is given by

$$QSred_i(cut) = 1 - \frac{Skew_i(Q, lb_i, cut) + Skew_i(Q, cut, ub_i)}{Skew_i(Q, lb_i, ub_i)} \tag{3}$$

where lb_i are ub_i are the lower and upper bounds of the node along dimension i and $lb_i < cut < ub_i$. Note that the number of candidates cuts along dimension i is $(SRN(ub_i) - SRN(lb_i) - 1)$ and the value for a candidate cut cut is the boundary value of two adjacent subranges. When $SRN(ub_i) - SRN(lb_i) = 1$, it means that along dimension i, the current node contains only one subrange and there is no candidate cut.

The algorithm for the construction of BT-Tree using CFBM, is presented in Algorithm 1. We first enqueue the root node that represents the whole data space into *queue* (line 3). Then nodes in *queue* are partitioned one by one until *queue* is empty (lines 4-17). Specifically, for each *node* that contains more than M trajectory segments, we identify cut_i, i.e., the cut that minimizes CF for each dimension i and add it to the set $CandCuts$ (lines 6-11). Among cuts in $CandCuts$, the one that achieves the smallest CF is the chosen cut, known as $FinalCut$ (lines 12-13). However, there is one special case to consider, i.e., when *node* covers only one subrange for every dimension. In this case, there is no candidate cut and $CandCuts$ is an empty set. Therefore, we consider three possible cuts, $medcut_i$ for $i \in \{x, y, t\}$ where $medcut_i$ is a cut at the median value along dimension i. $FinalCut$ is chosen to be the cut that achieves minimum percentage increase in the number of trajectory segments, $SegInc_i$ (lines 14-15). Then *node* is partitioned along $FinalCut$ and the two resulting child nodes are enqueued into *queue*.

Algorithm 1: Construction of BT-Tree (CFBM)

Input: Dataset D, query workload Q_t for each $t \in \{1, \ldots, type\}$, sets of subranges $SubR_i$ for each $i \in \{x, y, t\}$, vector $Hist_i(Q_t, a, b, n)$ for each $i \in \{x, y, t\}$, maximum block size M;
Output: BT-Tree;
Enqueue the BT-Tree root node into *queue*;
while *queue* **do**
  Pop the first element *node* of *queue*;
  if *node* overflows **then**
    $CandCuts \leftarrow \emptyset$;
    for $i \in \{x, y, t\}$ **do**
      if $h_i - l_i > 1$ **then**
        $cut_i = \text{argmin}_{cut}\, CF_i(cut)$;
        Add cut_i to $CandCuts$;
    if $CandCuts$ **then**
      $FinalCut \leftarrow cut$ that manimizes CF for $cut \in CandCuts$;
    else
      $FinalCut \leftarrow medcut_i$ with minimum $SegInc_i$;
    Partition *node* along $FinalCut$;
    Enqueue the 2 child nodes into *queue*;

4.3.4 Time Complexity. The time complexity of CFBM is $O(n^2/b)$ in the worst case, where n is the dataset size and b is the block size.

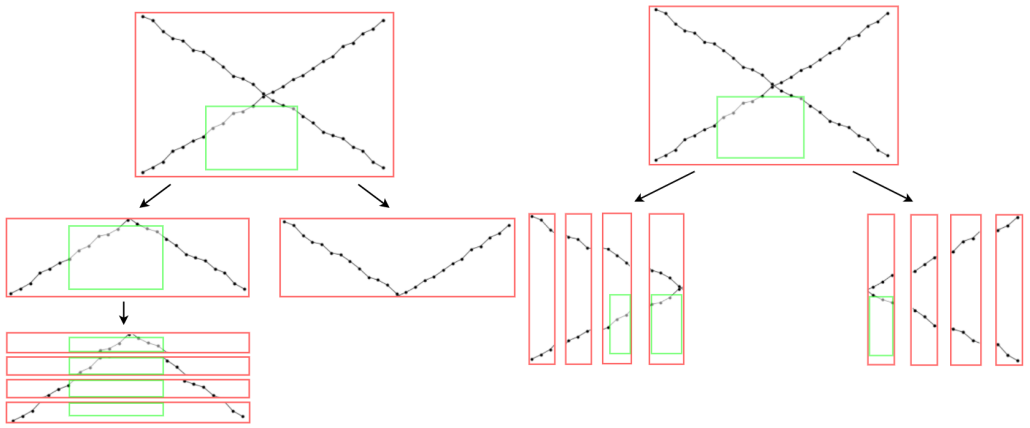
5 BT-Tree Construction (RL)

5.1 Motivation for RL

5.1.1 Limitations of CFBM. Firstly, CFBM relies on a cost function to partition a BT-Tree node. The parameter λ in the cost function has a significant impact on the query performance of the BT-Tree constructed (experimentally shown in Section 6.2.1). This is because λ reflects the relative impact of data-aware query skew reduction and the creation of new trajectory segments on the index performance. Secondly, CFBM always makes locally optimal choices which may lead to global suboptimality. To address the two limitations mentioned above, we propose an RL based method to construct BT-Tree.

5.1.2 Motivation for RL. Firstly, as explained above, it is difficult to decide what value of λ to set in the cost function. The application of RL enables us to learn the impacts of data-aware query skew reduction and the creation of new trajectory segments without involving λ. The high level idea is to reflect data-aware query skew reduction and the creation of new trajectory segments resulted from a cut (action) as parts of the state and then to learn their impacts by repeatedly building BT-Trees on a small training dataset and measuring their query performance so as to learn an optimal policy, which can then be used to build BT-Tree on big data.

Secondly, as mentioned above, CFBM always makes locally optimal choices which may lead to global suboptimality. Figure 6 shows a BT-Tree constructed by CFBM and a BT-Tree with better query performance for the given query q (in green). For clear illustration, we only show how tree nodes that intersect with query q are partitioned. The better BT-Tree has better query performance as q intersects with a smaller number of leaf nodes. However, when making the first partitioning decision at the root node, CFBM chooses the horizontal cut because the horizontal cut and not the vertical cut creates two child nodes, only one of which intersects q. When using an RL based method, we are enabled to build a complete tree first and then to evaluate each decision cut by



(a) BT-Tree Built by CFBM (b) A Better BT-Tree

Fig. 6. Motivation for Reinforcement Learning

measuring the goodness of the tree. Compared with CFBM, an RL based method is more likely to identify that the vertical cut is a better choice when making the first partitioning decision.

5.1.3 Challenges and Solutions. As we attempt to formulate this problem as a Markov Decision Process (MDP) and to use RL to choose a cut for an internal BT-Tree node, two main challenges have to be overcome to make it work. (1) *Formulation of the MDP.* Firstly, what constitutes the action space? A straightforward idea is to include all possible cuts in the action space. However, this approach leads to the formation of a large action space with many bad actions, especially for large nodes, which makes exploration during model training ineffective and inefficient. Our preliminary experimental results show that this simple design does not train the RL agent well. Our solution is to design the action space carefully and to use some hand-crafted rules to exclude "bad" cuts from the action space. Secondly, how to design the reward signal? It is important to design a good reward function to train the RL agent to take "good" actions. Since we aim to build a tree to process queries efficiently, the reward signal is expected to reflect the query performance. We propose to design the reward signal in a way that reveals if the current RL policy is better than some hand-crafted recursive bi-partitioning approach. Specifically, we consider an index built by recursively bi-partitioning the data space along alternate dimensions, which we call **BPAD**. The cut along a dimension is selected such that the two resulting partitions contain a similar number of trajectory segments. (2) *The large overhead incurred by model training.* Our preliminary experimental results show that, when we train the RL model on a large dataset (with size > 10 million), model training takes days and poor convergence is observed. It is therefore critical to find a solution that enables us to train the RL model on a small dataset and then to apply it on a large dataset. We propose to carefully design the state space, in order to train the model effectively and efficiently on a small dataset and then to be used on a large dataset. Since a node is an MBB represented by the 6 extreme values, a simple idea is to make the 6 extreme values be the state metrics. However, our preliminary experimental results show that this naive approach does not work well. One possible reason is that an MBB on the small dataset and a similar one on the large dataset may contain significantly different data (such as the number of trajectories) and hence have different optimal cuts. Therefore, the policy learned on the small dataset may not be readily applied on the large dataset. In our design, the state space captures the immediate impact of each action and the unique spatial features of the data. As the state metrics that capture the impact of an action are ratios (percentage values), they are likely to have similar values on the small dataset and

the large dataset. On top of that, as the spatial features of the data are captured, the RL model is enabled to learn the impact of each action in different regions of the data space. This explains why an RL model trained on the small dataset can be applied on the large dataset.

5.2 The RL Agent

We formulate this problem as a branching Markov Decision Process (MDP) [21] and use RL to choose a cut for an internal node. Next, we present the 4 key components of the MDP, namely *transitions*, *actions*, *states* and *rewards* and then the RL model training process.

5.2.1 Transitions. Given a state (a BT-Tree node) and an action (a cut), the RL agent transits to the two resulting child nodes. If a tree node contains no more than M trajectory segments, it is a leaf node and the RL agent reaches a terminal state.

5.2.2 Action Space. As explained in Section 5.1.3, the RL agent is not trained well when using a large action space and we propose to omit "bad" candidate cuts from the action space. Specifically, we consider the action space $\{a_x, a_y, a_t\}$ where a_i is the cut that achieves maximum CF_i along dimension i.

Remark. There may be no candidate cut along some dimension i, i.e., when $SRN(ub_i) - SRN(lb_i) = 1$. In this case, the corresponding action a_i in the action space is chosen to be $a_i = medcut_i$ where $medcut_i$ is the cut at the median value of the node along dimension i. Note that the data-aware query skew reduction of a_i in this case is 0. If there is no candidate cut along any dimension, we will not need RL and simply choose a cut from $\{medcut_x, medcut_y, medcut_t\}$ which causes the least increase in the number of trajectory segments.

5.2.3 State Space. As explained in Section 5.1.3, we first propose to design the state space in a way that captures the immediate impact of each candidate action, including the percentage data-aware query skew reduction and the percentage increase in the number of trajectory segments. For each action a_i in the action space, we compute the corresponding $QSred_i$ and $SegInc_i$ as its features, and then concatenate the features of the 3 actions in the action space to form a 6-dimensional vector. While this feature vector captures the immediate impact of candidate cuts on a node, it does not capture the unique spatial feature of each node, *i.e.*, the complex characteristics of the data region the node indexes. To address this issue, we add spatial embedding to the feature vector by mapping the boundaries between adjacent subranges into embedding vector u, where $u_{i,j}$ corresponds to the embedding of the j-th split point along each dimension i. We can obtain the embedding vector of a node's two boundary points along each dimension. The embedding vectors from all dimensions are then concatenated to form the spatial embedding of a state. Note that both $QSred_i$ and $SegInc_i$ are percentage values in the range of $[0, 1]$, and are not expected to differ significantly on the small training dataset and the large dataset, since the data distribution remains mostly the same. This enables us to train the RL mode on the small dataset and then to apply it on the large dataset. The spatial embedding further enables the RL model learn the spatial features and then to learn the impact of each candidate action in different parts of the data space. Our proposed state space significantly contributes to the reduction of model training time. In Section 6.2.7, we run experiments to evaluate the RL based methods with (**RL(SE)**) and without spatial embedding (**RL(NO SE)**).

5.2.4 Rewards. As explained in Section 5.1.3, the reward function is expected to reflect the difference in query performance between BT-Tree and BPAD. Therefore, after a BT-Tree is built (based on the current RL policy), we use the BT-Tree and BPAD built on the same dataset to process the queries in the historical query workload Q, and then compute and assign rewards to every state action pair. Note that BPAD is chosen as the competitor because BPAD and BT-Tree have similar tree structures. The rewards hence directly reflect whether or not the current policy achieves any improvement. Specifically, we compute the rewards through the following steps. (1) For each query

$q \in Q$, we compute IO_q^{BT} and IO_q^{KD} which are the I/O costs for the BT-Tree and BPAD to process q respectively. (2) For each internal node *node* in the BT-Tree with its respective state s and action a, we identify the set of queries $Q' \subseteq Q$ such that every query $q \in Q'$ intersects *node*. (3) The reward assigned to the state action pair (s, a) follows the following expression:

$$r(s,a) = 1 - \frac{\sum_{q \in Q'} IO_q^{BT}}{\sum_{q \in Q'} IO_q^{KD}}. \tag{4}$$

With the novel design of the reward signal, we are able to distinguish the good actions from the bad actions: A more positive reward means that the BT-Tree constructed using the current policy outperforms BPAD more significantly.

Algorithm 2: DQN Learning for BT-Tree

1 **Input:** Training dataset D, query workload Q_t for each $t \in \{1, \ldots, type\}$, sets of subranges $SubR_i$ for each $i \in \{x, y, t\}$, vector $Hist_i(Q_t, a, b, n)$ for each $i \in \{x, y, t\}$, maximum block size M;
2 **Output:** Learned action-value function $QN(s, a; \Theta)$;
3 Initialize $QN(s, a; \Theta)$, $\hat{QN}(s, a; \Theta^-)$;
4 Construct BPAD on D;
5 Initialize a BT-tree to be one with a root node only;
6 **for** $epoch = 1, 2, \ldots$ **do**
7 $Store \leftarrow \emptyset$;
8 Enqueue the BT-Tree root into *queue*;
9 **while** *queue* **do**
10 Pop the first element *node* of *queue*;
11 **if** *node* overflows **then**
12 $s \leftarrow$ state representation of *node*;
13 $a \leftarrow$ an action selected by ϵ-greedy based Q-values;
14 Add $(node, s, a)$ to $Store$;
15 Cut the node based on the action a (i.e., a cut) into 2 child nodes;
16 Enqeue the 2 child nodes to *queue*;
17 **for** $q \in Q$ **do**
18 Compute and store IO_q^{BT} and IO_q^{KD};
19 **for** each state action pair in $Store$ **do**
20 Compute *node*'s intersecting query set Q' and reward r;
21 Update $QN(; \Theta)$;
22 Periodically synchronize $\hat{QN}(; \Theta^-)$ with $QN(; \Theta)$;

5.2.5 Training the RL Agent Using DQN. We present the DQN [22] training algorithm for the construction of BT-Tree in Algorithm 2. We would like to highlight that the RL model is trained offline on a small subset of the given dataset. In order to facilitate the computation of data-aware query skew reduction, we follow Section 4 and compute Q_t, $SubR_i$ and $Hist_i(Q_t, a, b, n)$ in advance before model training starts. Note that the above computations can also be completed using the small training dataset if we do not have all the data available. We first initialize the main network $QN(s, a; \Theta)$ and the target network $\hat{QN}(s, a; \Theta^-)$ with the same random weights (line 3) and build BPAD based on the training dataset D (line 4). In each epoch, it first resets $Store$ which stores each internal node and its corresponding state action pair (line 7) and enqueues the BT-Tree root node into *queue* (line 8). For each overflowing node in *queue*, we compute the state representation (line 12) and use ϵ-greedy to choose the action based on their Q-values (line 13). We then add the node

and the corresponding state action pair to *Store* (line 14), partition the node using a (line 15) and then enqueue the two resulting child nodes to *queue* (line 16). After the tree is completely built, we use the BT-Tree and BPAD to process each query in Q and then compute and assign reward to each state action pair in *Store* based on the procedure in Section 5.2.4 (lines 17-20). Then we update the parameters in the main network $QN(;\Theta)$ (line 21) and synchronize $\hat{QN}$ with QN periodically (line 22).

5.2.6 Time Complexity. The time complexity of RL training is $O(e * n^2/b + e * f + e * |Q| * n/b)$ in the worst case, where n is the training data size, b is the block size, e is the number of training episodes, f denotes the cost of one neural network forward pass, and $|Q|$ is the size of the query workload.

5.3 Comparison with Existing Learned Indexes

Our proposed RL based method is fundamentally different from existing learned indexes. Existing learned indexes, such as Flood [23] and Tsunami [8], are designed for point data and learn the data distribution by explicitly learning a CDF based on the data points. They cannot be used to query trajectory data, where each trajectory is a sequence of points, rather than one multidimensional point, since existing CDF based learned indexes cannot be used to map a trajectory to a storage location. In contrast, our proposed RL based method learns an optimal policy to make partitioning decisions without directly learning the data distribution.

This approach works well for trajectory data because, (1) we can model the recursive bi-partitioning process as an MDP. Then by carefully designing the state space, the action space and the reward function, we can train the model on a small dataset and then use it on a large dataset. Our experiments (Figure 11) also show robust performance even when training and testing queries have different distributions; (2) we incorporate unique features of trajectory data (such as the creation of new segments resulted from a cut) into the model design such that an optimal policy can be effectively and efficiently learned. To the best of our knowledge, our work is the first that uses an RL based approach to index past trajectory data.

Remarks. Some learning based indexes are designed for dynamic environments where data updates occur frequently such as RLR-Tree [11], while some others are designed for static data such as Flood [23], Tsunami [8] and our proposed BT-Tree. As explained in Section 5.2.5, we train an RL model on a small dataset and then can use it to build a BT-Tree on a large dataset. This characteristic ensures the robustness of our proposed method even when there is new data that is not captured by the small training dataset. However, our proposed method is not designed to handle frequent data updates through insertion and deletion processes [11, 18]. When data updates occur, we may handle that through a periodic index rebuilding process. We leave more in-depth discussions of the dynamic environments to future work.

6 Experiments

6.1 Experimental Setup

Datasets. Our experiments are conducted on three real-life datasets. To ensure fair comparison, we choose similar datasets as those used in previous work, which we compare to as baselines (e.g., [6, 48]). The first dataset, Geolife, records outdoor trajectories of 182 users for a period of 5 years. The second dataset, Porto, contains more than 1.7 millions taxi trajectories in Porto. The third dataset, T-Drive, tracks the trajectories of 10,357 taxis in Beijing. The largest dataset considered in our experiments contains 800 million points.

Queries. We consider range queries of different sizes for the experiments. Note that we follow previous work [28–30, 48] on spatial and trajectory data indexing and assume that the queries

and the data follow a similar distribution. Specifically, one workload contains 2,000 range queries, each of which covers ss of each spatial dimension and ts of the temporal dimension of the whole data space. We follow the settings in previous work [6, 48] and consider $ts \in \{0.1\%, 1\%, 10\%\}$ and $ss \in \{0.1\%, 1\%, 10\%\}$. A range query is generated by randomly selecting a point from the dataset and set it to be the center of the query. For Porto and T-Drive, this point is set to be the starting or the ending point of a randomly selected trajectory, which is likely to be a taxi stand. Then its length along each dimension follows the values of ts and ss accordingly. We also consider workload changes by considering range queries of different sizes and different distributions and KNN queries, in order to test the robustness of our proposed methods.

Compared methods. First of all, we test our proposed non-RL method (**CFBM**) (Section 4) and the RL method (**RL(SE)**) that incorporates spatial embedding in the state design (Section 5). Besides that, we also consider a greedy method (**MinNode**) to build the BT-Tree. Specifically, an internal tree node is partitioned by the cut that minimizes the number of queries that intersect with resulting child nodes among all candidates cuts. We also consider **RL(SE)_Tsu**, which incorporates Tsunami's approach to handle query skew into our proposed RL framework. (Detailed discussions on Tsunami's approach to handle query skew can be found in Sections 2.3, 4.1 and 4.3.2.) Finally, baseline methods we compare to include **TB-Tree** [26], **TrajStore** [6] and **GCOTraj** [48], which represent the state of the art indexes for past trajectory data, and **SQUID** [52], which uses a temporal first approach to index trajectory data. Specifically, it builds temporal partitions at the top followed by an R-Tree for each temporal partition at the bottom. The implementation details of the baseline methods are as follows. (1) TB-Tree: we maintain a maximum of 100 entries per tree node. (2) GCOTraj: we use the version of GCOTraj that has the best query performance in the paper, i.e., 3D GCOTraj_GBO, with the grid size set to be 64. (3) TrajStore: we use the version of TrajStore that has the best query performance and set the maximum level of the quad-tree to be 6. This setting is consistent with the setting in [48]. (4) SQUID: we set the number of partitions along the temporal dimension to be 128 and maintain a maximum of 100 entries per node in each R-Tree.

Measurements. For measurements of query performance, we consider both running time and the I/O cost. We find that both measures yield qualitatively consistent results (as exemplified in Figure 7 and 8). We follow [18] to report the average relative I/O cost mainly. For each query, the relative I/O cost of an index is computed by the ratio of the I/O cost for it to answer the query to the I/O cost for BPAD (Section 5.1.3) to answer the same query. Smaller relative I/O costs indicate better query performance than BPAD. BPAD is chosen as the reference because both BT-Tree and BPAD are constructed by recursive bi-partitioning approaches. It is therefore easy to observe the improvement our proposed method achieves. Note that this choice does not affect the reporting of the experimental results as all the baselines are eventually compared with.

Table 2. Parameters and Values

Parameters	**Values**
Dataset	**Geolife**, Porto, T-Drive
Temporal predicate size (%)	0.1, **1**, 10
Spatial predicate size (%)	0.1, **1**, 10
Training dataset size (thousand)	10, **100**, 200, 300, 400

Parameter settings. Table 2 shows a list of parameters and their corresponding values tested in our experiments. The default settings are bold. For all compared methods in this paper, we maintain a maximum of 100 trajectory segments per page. When building a BT-Tree, we set the total number of subranges n along each dimension to be 128. (Our preliminary experiments show that a smaller n value causes the model performance to deteriorate while a larger n value causes the computation

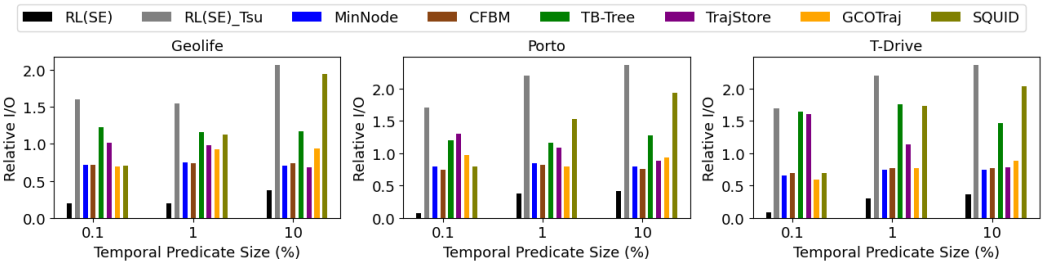


Fig. 7. Range Query Performance with Changing Temporal Predicate Sizes

cost to increase significantly without improving the model performance. Due to space constraint, we do not report the details of the experiments.) The DQN model contains 1 hidden layer of 64 neurons with SELU [13] as the activation function. In the training process, the learning rate is set to be 0.003. The value of ϵ is set to be 0.1 so as to to maintain a certain degree of exploration throughout model training. Network update and synchronization is done at the end of each epoch. We train the RL model for a total of 40 epochs. In each epoch, we use the RL model to build a complete BT-Tree on a small training dataset.

We train our models on NVIDIA Tesla V100 SXM2 16 GB GPU using PyTorch 1.3.1. All indexes are coded using C++.

6.2 Experimental Results

Our experiments aim to find out: 1) what value of λ is optimal for the cost function in the CFBM method (Section 6.2.1); 2) how well our proposed methods perform compared with baselines for range queries (Section 6.2.2); 3) how well our proposed methods perform compared with baselines for KNN queries (Section 6.2.3); 4) how well our proposed methods perform when query workload changes (Section 6.2.4); 5) how the model performance changes when the training dataset size changes (Section 6.2.6); 6) how the model performance changes when using different RL model designs (Section 6.2.7); 7) how well our proposed methods scale with dataset sizes (Section 6.2.8); and 8) the index size of our proposed methods (Section 6.2.9).

Table 3. Performance (Relative I/O) of the CFBM

	$\lambda = 0$	$\lambda = 0.25$	$\lambda = 0.5$	$\lambda = 0.75$	$\lambda = 1$
Geolife	0.95	0.75	**0.74**	1.22	3.55
Porto	0.99	0.77	**0.72**	1.55	2.89
T-Drive	1.05	**0.80**	0.84	2.38	3.01

6.2.1 Evaluation of Parameter λ in CFBM. In this set of experiments, we evaluate the impact of λ in the cost function of CFBM. In Table 3, we use CFBM with different λ values to build BT-Trees and report the average relative I/O cost when processing queries from the default workload, i.e., temporal predicate size = 1% and spatial predicate size = 1%. We observe that on different datasets, the optimal values for λ are not the same ($\lambda = 0.5$ for Geolife and Porto, $\lambda = 0.25$ for T-Drive), although the difference between $\lambda = 0.5$ and $\lambda = 0.25$ is generally insignificant. We would like to highlight that when $\lambda = 0$ or $\lambda = 1$, the corresponding BT-Trees show rather poor query performances. It shows that both data-aware query skew reduction and the creation of new trajectory segments resulted from a cut have an impact on the performance of BT-Tree.

6.2.2 Range Queries. In this set of experiments, we evaluate the performance of different methods on range queries with different spatial and temporal predicate sizes.

Temporal Predicate Size. Figure 7 reports the average relative I/O cost of all compared methods on all 3 datasets as the size of the temporal predicate changes. We observe that our best proposed

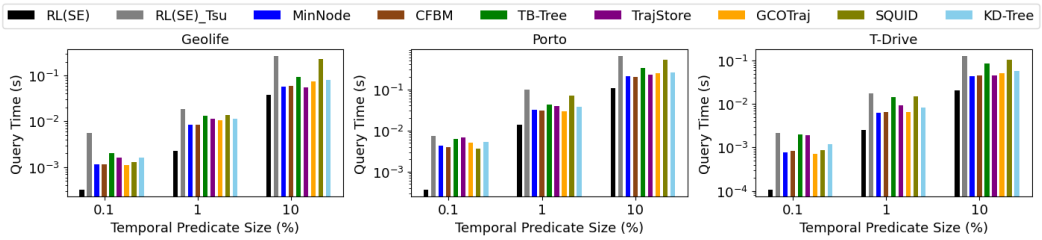


Fig. 8. Range Query Performance with Changing Temporal Predicate Sizes (Query Time)

method, RL(SE) consistently has the best query performance. It is remarkable that although the training dataset size is only 100,000, which is less than 1% of the size of the complete dataset, the trained model can be applied on large datasets successfully. Moreover, RL(SE)'s advantage gets increasingly significant as the temporal predicate size decreases. This is because a query with a larger size usually intersects with a larger number of tree nodes, and hence more tree nodes will be accessed when answering it. In this case, every index will visit a larger proportion of the data and relative I/O will hence be to closer to 1. We would like to highlight that RL(SE) outperforms CFBM significantly, indicating the existence of a gap between local optimality and global optimality, as discussed in Section 5.1.2. One possible reason is that, when indexing large datasets, the large tree depth makes locally optimal partitioning decisions (especially at tree nodes that are located near to the root) no longer as effective to ensure the quality of the final tree structure. However, with our proposed RL based method, we are able to train an RL model by building complete trees (on a small training dataset) and then evaluating each partitioning decision by measuring the goodness of the trees, and then to use the model to build an index tree (on a large dataset). Although our proposed non-RL based methods, MinNode and CFBM, that leverage on the query workload to build BT-Tree, are consistently outperformed by RL(SE), they outperform the baselines, TB-Tree, TrajStore, GCOTraj and SQUID, most of the time. RL(SE)_Tsu consistently shows poor query performances, mainly because of its two major limitations (as discussed in Sections 2.3, 4.1 and 4.3.2), i.e., (1) it takes into consideration query distribution and not data distribution; and (2) it does not distinguish queries that have similar sizes but cover different parts of the data space resulting in inaccurate computation of query skew.

Among TB-Tree, TrajStore, GCOTraj and SQUID, TB-Tree consistently has the worst query performance. This result is expected because TB-Tree only allows segments from the same trajectory to be stored on the same leaf nodes. With this approach, segments that are spatially or/and temporally close to each other are not always stored on the same node which then compromises the pruning effect of the index when answering range queries. TrajStore's query performance deteriorates as the temporal predicate gets more selective. Similar results are also reported in [6]. This trend can be attributed to the index design of TrajStore. As explained in Section 2.1, as TrajStore builds a spatial index at the top, it is rather ineffective to prune irrelevant data when the temporal predicate is highly selective. Specifically, when the temporal predicate size is 0.1%, RL(SE) outperforms TrajStore by up to 94.6%. SQUID's query performance deteriorates as the temporal predicate gets less selective. This is because the temporal first indexing approach adopted by SQUID does not prune irrelevant data effectively when the temporal predicate is not selective. For GCOTraj, although we follow [48] and use the optimal setting, it is still outperformed by RL(SE) by up to 92.8%. One possible reason is that GCOTraj's grid size is independent of the query workload. For example, when the queries are not selective along a particular dimension, retaining the same grid size along the dimension will lead to additional I/O cost in query processing because having too many partitions along the dimension does not help to prune irrelevant data effectively.

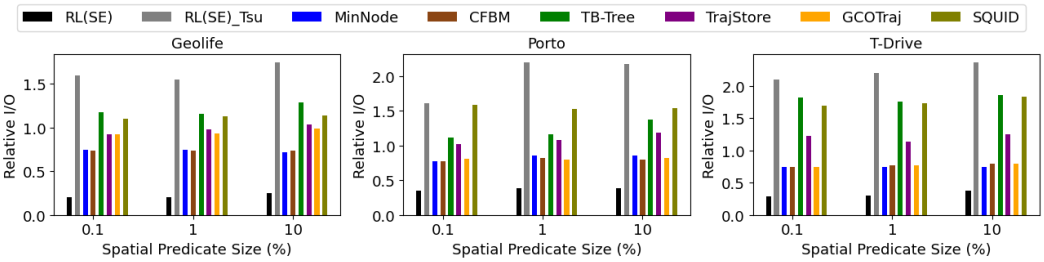


Fig. 9. Range Query Performance with Changing Spatial Predicate Sizes

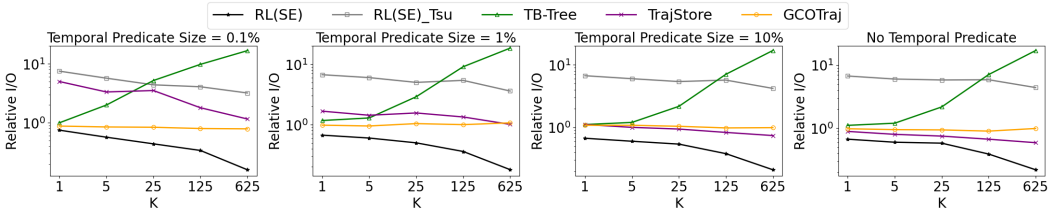


Fig. 10. KNN Query Performance

Figure 8 reports the average query time of all compared methods on all 3 datasets for different temporal predicate sizes. We would like to highlight that the ratio of the query time of any method to that of BPAD is generally consistent with the relative I/O cost reported in Figure 7. As a result, for the remaining of the paper, we follow previous work such as [18] and report only relative I/O cost as a measure of query performance.

Spatial Predicate Size. Figure 9 reports the average relative I/O cost of all compared methods on all 3 datasets as the size of the temporal predicate changes. We observe that RL(SE) always has the best performance. At the same time, RL(SE)'s advantage gets increasingly significant as the spatial predicate size decreases. This trend is consistent with that when the temporal predicate size decreases and possible reasons have been discussed above.

6.2.3 KNN Queries. We would like to evaluate our best proposed method, RL(SE), for a type of queries that is not used in the model training process, i.e., the K-Nearest-Neighbor (KNN) queries. A KNN query returns the K nearest objects to a given query point. We follow the definition of point-to-trajectory distance in [42] to measure the distance between a trajectory segment to a query point. Furthermore, the KNN queries in this context return the K spatially nearest objects in a given temporal range. When processing a KNN query, we use the same query processing algorithm as [11] and always prune tree nodes that are out of the temporal range of the query. In our experiments, we consider different K values from $\{1, 5, 25, 125, 625\}$ and different temporal predicate sizes from $\{0.1\%, 1\%, 10\%\}$. For each K value and temporal predicate size, 2,000 query points are randomly generated in the spatial space and for each query point, the center of the corresponding temporal range is randomly generated in the temporal space. Note that when processing KNN queries, any tree node that does not intersect with the given temporal range is pruned.

Figure 10 reports the relative I/O cost of our best proposed method, i.e., RL(SE), and the three baselines i.e., TB-Tree, TrajStore and GCOTraj. We also compare with RL(SE)_Tsu. Due to space constraint, we only show the results on the Geolife dataset. Note that the experimental results on all three datasets are qualitatively similar. We observe that RL(SE) consistently outperforms the baselines for all values of K and all temporal predicate sizes, and has a larger advantage for larger K values. Specifically, RL(SE) outperforms the best baseline by up to 82.2%. It is remarkable that RL(SE) performs well on KNN queries, although the RL model is designed and trained to optimize the query performance for range queries. One possible reason is that, although KNN queries and

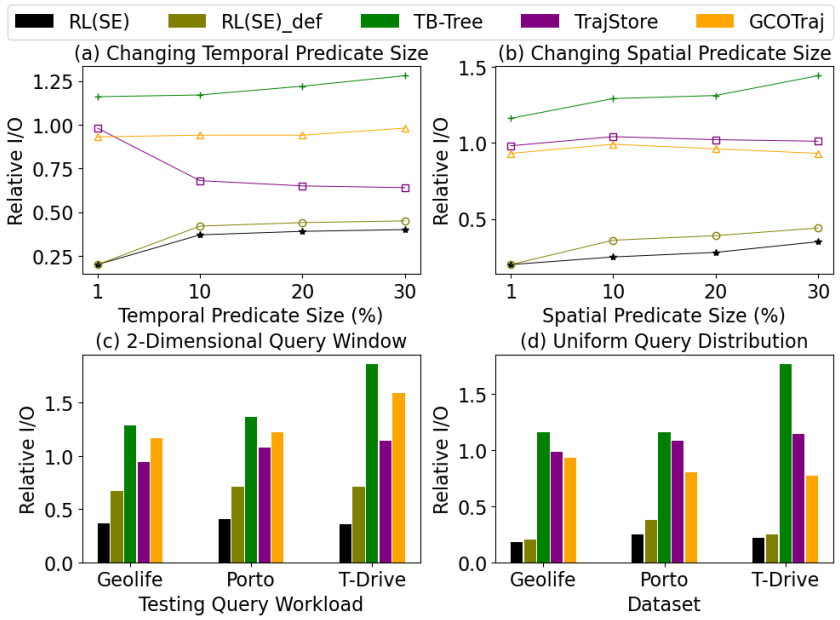


Fig. 11. Effects of Changing Query Workload

range queries are different, they share certain similarities in their pruning processes, i.e., to discard entries that are far from the query region/point. As a result, a model trained using range queries can handle KNN queries reasonably well. RL(SE)_Tsu consistently shows significantly poorer query performance than RL(SE). This result on KNN quries is consistent with that for range queries. Among the baselines, TB-Tree's performance deteriorates significantly as the value of K increases and is up to 20 times worse than BPAD when $K = 625$. This is because in a TB-Tree, trajectory segments that are spatially close to each other but are not from the same trajectory are not stored on the same leaf node. As a result, its spatial pruning gets increasingly ineffective as the value of K increases. As K increases, TrajStore's performance improves because of the effectiveness of it is spatial index. When there is no temporal predicate, Trajstore has the best performance among all baselines. Last but not least, we observe that GCOTraj has similar query performance compared to BPAD.

6.2.4 Changing Query Workload. In this experiment, we would like to test how well our proposed method adapts to query change and to evaluate the performance of RL(SE) when it is trained using one query workload and tested on a different query workload.

Changing Query Sizes. Using the default Geolife dataset, we first train an RL(SE) model using range queries with the default query size and then use the trained model to build an BT-Tree to process queries of different sizes. In Figures 11a and 11b, we vary the spatial and the temporal predicate sizes respectively and report the average relative I/O cost. Note that in this set of experiments, "RL(SE)_def" represents the query performance of this BT-Tree while "RL(SE)" represents the performance of the model trained and tested on the same query workload. We observe that RL(SE)_def consistently outperforms the baselines although its performance is slightly poorer than RL(SE).

Changing Query Dimension. Recall (Section 6.1) that the range queries used to train the RL models are 3-dimensional queries. In this experiment, we first train an RL(SE) model using this workload and then test the model on 2-dimensional range queries. Each 2-dimensional range query consists of a randomly selected spatial dimension (i.e., the selection on the other spatial dimension is

Table 4. Performance (Relative I/O) of RL(SE)

	Geolife (Tr)	Porto (Tr)	T-Drive (Tr)
Geolife (Test)	**0.20**	0.45	0.54
Porto (Test)	0.79	**0.38**	0.52
T-Drive (Test)	0.85	0.55	**0.30**

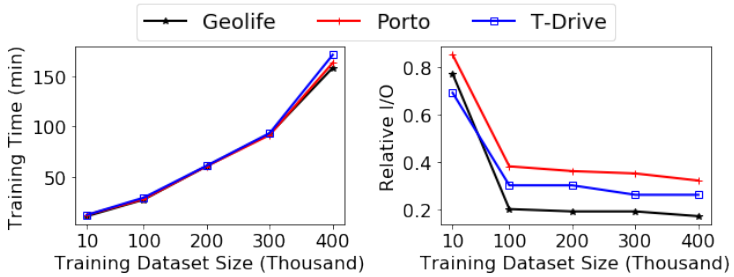


Fig. 12. Effect of Varying Training Dataset Sizes

100%) and the temporal dimension and its size follows the default setting. In Figure 11c, "RL(SE)_def" represents the query performance of this BT-Tree while "RL(SE)" represents the performance of the model trained and tested on 2-dimensional range queries. We observe that RL(SE)_def consistently outperforms the baselines although RL(SE) outperforms RL(SE)_def by up to 44%.

Changing Query Distribution. Recall (Section 6.1) that the query workload used in RL training follows the distribution of the datasets. In this experiment, we first train an RL(SE) model using this workload and then use the trained model to build a BT-Tree to process a uniformly distributed workload. In Figure 11d, "RL(SE)_def" represents the query performance of this BT-Tree while "RL(SE)" represents the performance of the model trained and tested on a uniformly distributed workload. We observe that despite the change in query distribution, RL(SE)_def consistently outperforms the baselines. Note that RL(SE) outperforms RL(SE)_def slightly.

Remarks. The experimental results show that our proposed method consistently outperforms the baselines even when the characteristics of the training workload and that of the testing workload are different. One key reason behind the robustness of our proposed method is that the spatial embedding enables the RL model to learn important characteristics of the dataset which remains useful despite query change. Even when the characteristics of the testing queries are different, the learned policy remains relevant to a certain extent, although the model performance deteriorates compared with a model trained and tested on the same workload.

6.2.5 Changing Data Distribution. This experiment is to evaluate the robustness of our proposed method by training the RL(SE) model on one data distribution and then using it on a different data distribution. As shown in Table 4, the model performance is the best when it is trained and tested on the same data distribution. When the RL(SE) model is trained on one data distribution and tested on a different distribution, the model performance expectedly deteriorates but still achieves some improvement compared to BPAD. One possible reason for the resilient performance is that some of the knowledge initially learned by the RL model remains useful even if the data distributions are changed. This is also one key advantage of our method compared to CDF based methods.

6.2.6 Changing Training Dataset Size. This experiment is to evaluate the effect of training dataset size on the performance of the RL(SE) model. The results in Figure 12 show that we can train the model on a small dataset and then apply it on a large dataset. It significantly contributes to the reduction of overhead incurred by model training. When the training dataset size is 10,000, RL(SE) is only able to achieve up to 30% of performance improvement compared to BPAD although training time is only 10 minutes. As we use larger training datasets, the model training time increases at

an increasing rate and shows similar trends on all 3 datasets. However, model performance does not improve significantly as the training dataset size increases from 100,000 to 400,000. As a result, we set 100,000 to be the default size for the training dataset as it achieves a good balance between training time and performance.

6.2.7 Changing RL Model Designs.

In this set of experiments, we conduct an ablation study to find out how much improvement in model performance is achieved as we incorporate data property and query property into the design of the RL model, respectively.

Data Property. As explained in Section 5.2.3, our design of the state space captures both the immediate impact of candidate actions at a tree node and the unique spatial feature of the node. As shown in Table 5, when the state only captures the immediate impact of candidate actions, the RL model achieves significant performance improvement. However, when data property is incorporated in the design of the state space of the RL model, i.e., spatial embedding is included to reflect the spatial features of an internal node, the RL model achieves additional performance improvement of up to 69%.

Table 5. Impact of spatial embedding on the RL model

	Geolife	**Porto**	**T-Drive**
RL(SE)	0.20	0.38	0.30
RL(NO SE)	0.65	0.62	0.47

Query Property. As explained in Section 5.2.3, the query skew reduction ($QSred_i$) achieved by each candidate action (a_i) forms part of our designed state space. As shown in Table 6, as we exclude and include this query property in the state space (represented by RL(NO QS) and RL(SE) respectively), the model performance improves by up to 56%.

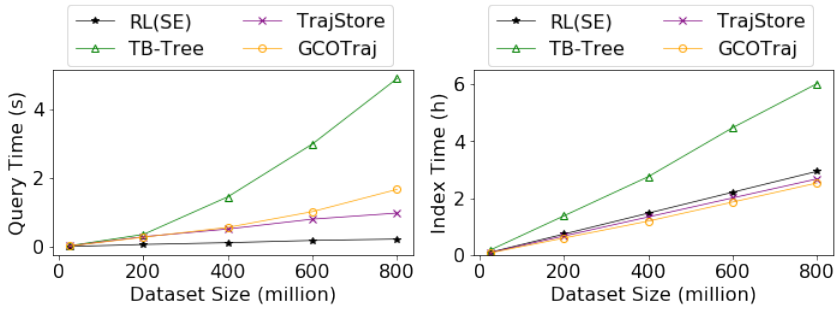
Table 6. Impact of query skew on the RL model

	Geolife	**Porto**	**T-Drive**
RL(SE)	0.20	0.38	0.30
RL(NO QS)	0.45	0.72	0.67

6.2.8 Scalability Test. In this experiment, we test the robustness of our proposed method on larger datasets of size up to 800 million using the default query workload, i.e., temporal predicate size = 1% and spatial predicate size = 1%. We use the original Geolife dataset to create larger datasets by repeatedly making duplicates of all trajectories and then shifting the duplicate trajectories by sp_i% of the data space along each dimension i, where sp_i is a random number in the range $[-1, 1]$. Note that the largest dataset size of about 800 million is obtained by repeating the above step 5 times.

Query Time. Figure 13a reports the query time as dataset size increases. We make a few observations. Firstly, TB-Tree consistently has the poorest performance. Secondly, GCOTraj's query time increases at an increasing rate as dataset size increases. This is possibly because GCOTraj prunes irrelevant data less effectively on larger datasets. Finally, TrajStore's and RL(SE)'s query time generally increases linearly with dataset size. We would like to highlight that RL(SE)'s advantage gets more significant on larger datasets.

Index Construction Time. Figure 13b reports the index construction time as dataset size increases. Among the compared methods, TB-Tree has the longest construction time while RL(SE), TrajStore and GCOTraj have similar construction time. All 4 methods show linear increases in construction time with dataset size.



(a) Query Time (b) Index Construction Time

Fig. 13. Scalability Test

Table 7. Index Size (GB)

RL(SE)	MinNode	CFBM	TB-Tree	TrajStore	GCOTraj
22.9	26.6	26.1	19.6	23.8	28.3

6.2.9 Index Size. In this section, we look into the index size of each method. We follow Section 6.2.8 and use the largest dataset of size 800 million. As shown in Table 7, we observe that TB-Tree's size is the smallest because it is an R-Tree based method and creates no additional trajectory segment. Our RL based indexes have smaller index size than MinNode, CFBM and GCOTraj. This is probably because the RL models are able to learn that, cuts which create a large number of additional trajectory segments tend to have a negative impact on the query performance of the index, and then avoid these cuts and build smaller indexes than other methods.

7 Conclusions and Future Work

We propose a novel index, BT-Tree, to index past trajectory data, based on historical query workload. To construct BT-Tree, we first propose a greedy method that relies on a cost function that captures data-aware query skew reduction and the increase in the number of trajectory segments resulted from space partitioning. Subsequently, we propose an RL based method to address the limitations of the greedy method. Experimental results show that our RL based method consistently outperforms the baselines for both range and KNN queries. We would like to highlight that our proposed method has especially remarkable performance on larger datasets and shows robust performance even when a trained RL model is used on queries of different sizes and distributions.

This work takes a first step to use machine learning to build trajectory data indexes, and would open new directions for future work: 1) extend the idea to index moving objects (with velocities) for the prediction of future locations; and 2) explore the possibility to apply machine learning on similarity searches for trajectory data.

8 Acknowledgements

This research is supported in part by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Awards MOE-T2EP20221-0015, MOE-T2EP20223-0004, MOE-T2EP20221-0013, MOE-T2EP20220-0011). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. This research is supported in part by grant No. 62394333 from the National Natural Science Foundation of China.

References

[1] Pankaj K Agarwal, Jie Gao, and Leonidas J Guibas. 2002. Kinetic medians and kd-trees. In *Algorithms—ESA 2002: 10th Annual European Symposium Rome, Italy, September 17–21, 2002 Proceedings 10*. Springer, 5–17.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. 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*. 322–331.

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[4] V Prasad Chakka, Adam Everspaugh, Jignesh M Patel, et al. 2003. Indexing large trajectory data sets with SETI.. In *CIDR*, Vol. 75. Citeseer, 76.

[5] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A Nascimento. 2008. ST2B-tree: a self-tunable spatio-temporal B+-tree index for moving objects. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 29–42.

[6] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. 2010. Trajstore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 109–120.

[7] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.. In *EDBT*. 407–410.

[8] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

[10] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 247–252.

[11] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[12] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[13] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-normalizing neural networks. *Advances in neural information processing systems* 30 (2017), 971–980.

[14] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[15] Hai Lan, Jiong Xie, Zhifeng Bao, Feifei Li, Wei Tian, Fang Wang, Sheng Wang, and Ailin Zhang. 2022. VRE: a versatile, robust, and economical trajectory data system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3398–3410.

[16] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th International Conference on Data Engineering*. IEEE, 497–506.

[17] Jiangneng Li, Zheng Wang, Gao Cong, Cheng Long, Han Mao Kiah, and Bin Cui. 2023. Towards designing and learning piecewise space-filling curves. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2158–2171.

[18] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.

[19] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. Trajmesa: A distributed nosql storage engine for big trajectory data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2002–2005.

[20] ZhengYu Li and ZhuoFeng Zhao. 2021. MGeohash: Trajectory data index method based on historical data pre-partitioning. In *2021 7th International Conference on Big Data Computing and Communications (BigCom)*. IEEE, 241–246.

[21] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*. 256–269.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[23] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 985–1000.

[24] Thi Nguyen, Zhen He, Rui Zhang, and Phillip Ward. 2012. Boosting moving object indexing through velocity partitioning. *Proceedings of the VLDB Endowment* 5, 9 (2012), 860–871.

[25] Mindaugas Pelanis, Simonas Šaltenis, and Christian S Jensen. 2006. Indexing the past, present, and anticipated future positions of moving objects. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 255–298.

[26] Dieter Pfoser, Christian S Jensen, Yannis Theodoridis, et al. 2000. Novel approaches to the indexing of moving object trajectories.. In *VLDB*. Citeseer, 395–406.

[27] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. 2010. PARINET: A tunable access method for in-network trajectories. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 177–188.

[28] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[29] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically optimal and empirically efficient r-trees with strong parallelizability. *Proceedings of the VLDB Endowment* 11, 5 (2018), 621–634.

[30] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2020. Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–47.

[31] Hani Ramadhan and Joonho Kwon. 2022. X-FIST: Extended flood index for efficient similarity search in massive trajectory dataset. *Information Sciences* 606 (2022), 549–572.

[32] Simonas Šaltenis, Christian S Jensen, Scott T Leutenegger, and Mario A Lopez. 2000. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 331–342.

[33] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. 2011. Indexing in-network trajectory flows. *The VLDB Journal* 20, 5 (2011), 643–669.

[34] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: distributed in-memory trajectory analytics. In *Proceedings of the 2018 International Conference on Management of Data*. 725–740.

[35] Na Ta, Guoliang Li, Yongqing Xie, Changqi Li, Shuang Hao, and Jianhua Feng. 2017. Signature-based trajectory similarity join. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2017), 870–883.

[36] Yufei Tao and Dimitris Papadias. 2001. The mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome*.

[37] Yufei Tao, Dimitris Papadias, and Jimeng Sun. 2003. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings 2003 VLDB conference*. Elsevier, 790–801.

[38] Yannis Theodoridis, Michael Vazirgiannis, and Timos Sellis. 1996. Spatio-temporal indexing for large multimedia applications. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*. IEEE, 441–448.

[39] Ruijie Tian, Weishi Zhang, Fei Wang, Kemal Polat, and Fayadh Alenezi. 2023. Tinba: Incremental partitioning for efficient trajectory analytics. *Advanced Engineering Informatics* 57 (2023), 102064.

[40] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 569–574.

[41] Longhao Wang, Yu Zheng, Xing Xie, and Wei-Ying Ma. 2008. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. In *The Ninth International Conference on Mobile Data Management (mdm 2008)*. IEEE, 1–8.

[42] Sheng Wang, Zhifeng Bao, J Shane Culpepper, and Gao Cong. 2021. A survey on trajectory data management, analytics, and learning. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[43] Sheng Wang, Zhifeng Bao, J Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. 2018. Torch: A search engine for trajectory data. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 535–544.

[44] Shuang Wang and Hakan Ferhatosmanoglu. 2020. PPQ-trajectory: spatio-temporal quantization for querying in large trajectory repositories. *Proceedings of the VLDB Endowment* 14, 2 (2020), 215–227.

[45] Dong Xie, Feifei Li, and Jeff M Phillips. 2017. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1478–1489.

[46] Dong Xie, Feifei Li, and Jeff M Phillips. 2017. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1478–1489.

[47] Jingyi Yang and Gao Cong. 2023. PLATON: Top-down R-tree Packing with Learned Partition Policy. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[48] Shengxun Yang, Zhen He, and Yi-Ping Phoebe Chen. 2018. GCOTraj: A storage approach for historical trajectory data sets using grid cells ordering. *Information Sciences* 459 (2018), 1–19.

[49] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[50] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. 2008. The B dual-Tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal* 17, 3 (2008), 379–400.

[51] Haitao Yuan, Guoliang Li, Zhifeng Bao, and Ling Feng. 2020. Effective travel time estimation: When historical trajectories over road networks matter. In *Proceedings of the 2020 acm sigmod international conference on management of data*. 2135–2149.

[52] Dongxiang Zhang, Zhihao Chang, Dingyu Yang, Dongsheng Li, Kian-Lee Tan, Ke Chen, and Gang Chen. 2023. SQUID: subtrajectory query in trillion-scale GPS database. *The VLDB Journal* 32, 4 (2023), 887–904.

[53] Rui Zhang, H. V. Jagadish, Bing Tian Dai, and Kotagiri Ramamohanarao. 2010. Optimized algorithms for predictive range and KNN queries on moving objects. *Inf. Syst.* 35, 8 (2010), 911–932. https://doi.org/10.1016/J.IS.2010.05.004

[54] Rui Zhang, Beng Chin Ooi, and Kian-Lee Tan. 2004. Making the Pyramid Technique Robust to Query Types and Workloads. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, Z. Meral Özsoyoglu and Stanley B. Zdonik (Eds.). IEEE Computer Society, 313–324. https://doi.org/10.1109/ICDE.2004.1320007

[55] Yue Zhao, Yunhai Wang, Jian Zhang, Chi-Wing Fu, Mingliang Xu, and Dominik Moritz. 2021. KD-Box: Line-segment-based KD-tree for interactive exploration of large-scale time-series data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 890–900.

Received January 2024; revised April 2024; accepted May 2024