Article

A Time-Identified R-Tree: A Workload-Controllable Dynamic Spatio-Temporal Index Scheme for Streaming Processing

Weichen Peng [1], Luo Chen [1,2,*], Xue Ouyang [1] and Wei Xiong [1]

1 College of Electronic Science and Technology, National University of Defense Technology, Changsha 410073, China; pengweichen@nudt.edu.cn (W.P.); ouyangxue08@nudt.edu.cn (X.O.); xiongwei@nudt.edu.cn (W.X.)
2 Key Laboratory of Natural Resource Surveying and Monitoring of Southern Hilly Area, MNR, Changsha 410118, China
* Correspondence: luochen@nudt.edu.cn; Tel.: +86-139-7580-1542

Abstract: Many kinds of spatio-temporal data in our daily lives, such as the trajectory data of moving objects, stream natively. Streaming systems exhibit significant advantages in processing streaming data due to their distributed architecture, high throughput, and real-time performance. The use of streaming processing techniques for spatio-temporal data applications is a promising research direction. However, due to the strong dynamic nature of data in streaming processing systems, traditional spatio-temporal indexing techniques based on relatively static data cannot be used directly in stream-processing environments. It is necessary to study and design new spatio-temporal indexing strategies. Hence, we propose a workload-controllable dynamic spatio-temporal index based on the R-tree. In order to restrict memory usage, we formulate an INSERT and batch-REMOVE (I&BR) method and append a collection mechanism to the traditional R-tree. To improve the updating performance, we propose a time-identified R-tree (TIR). Moreover, we propose a distributed system prototype called a time-identified R-tree farm (TIRF). Experiments show that the TIR could work in a scenario with a controllable usage of memory and a stable response time. The throughput of the TIRF could reach 1 million points per second. The performance of a range search in the TIRF is many times better than in PostgreSQL, which is a widely used database system for spatio-temporal applications.

Keywords: R-tree; streaming processing; spatio-temporal index

check for updates

Citation: Peng, W.; Chen, L.; Ouyang, X.; Xiong, W. A Time-Identified R-Tree: A Workload-Controllable Dynamic Spatio-Temporal Index Scheme for Streaming Processing. *ISPRS Int. J. Geo-Inf.* **2024**, *13*, 49. https://doi.org/10.3390/ijgi13020049

Academic Editors: Zhihua Zhang, M. James C. Crabbe and Wolfgang Kainz

Received: 22 November 2023
Revised: 25 January 2024
Accepted: 31 January 2024
Published: 4 February 2024

1. Introduction

Techniques of streaming processing have developed over these years. Engines that handle streaming processing tasks, for example, Spark Streaming (https://spark.apache.org/streaming/, accessed on 12 October 2023) and Flink (https://flink.apache.org/, accessed on 12 October 2023), are an essential support of many business applications, such as fraud detection, ad hoc analysis, rules-based alerting, and business-process monitoring.

There is a kind of data stream that we cannot ignore. With the development of communication techniques, sensors, most commonly mobile devices with GPS functionality, generate spatio-temporal (ST) messages, such as driving routes for cars, sign-in records in APPS, trajectories of flights, and location information in navigation, then send them to servers continuously in real time. Such a large-scale stream is received, processed, and stored in low latency. It poses a challenge to index and manage the ST data stream.

In traditional spatio-temporal indexing tasks, there are several mature solutions. In the widely used relational database PostgreSQL (https://www.postgresql.org/, accessed on 12 October 2023), an extension named PostGIS provides spatio-temporal operators. After creating a dynamic ST index called a GiST (generalized search tree, a replacement of the R-tree in PostGIS), all the ST operations are accelerated. Another example is GeoMesa (https://www.geomesa.org/, accessed on 12 October 2023). Massive ST data in GeoMesa are managed by a spacing-filling curve, the Z curve, which is a type of static index method.

Our study focuses on the dynamic index. Based on our knowledge, a dynamic index could have a better adaptability for drifts in stream data than a static index, since it fits in well with the data distribution. However, the building cost of a dynamic index is much greater, especially when the data scale is large and updates occur frequently. Since we are dealing with a data stream, it is an option to solve the dynamic indexing problem by streaming methods.

Streaming processing has the following properties. The data elements arrive online from multiple sources sequentially, like water streams, while the system cannot control the order and the arrival time. The data stream is almost infinite as long as the system is running. To make sure every element is addressed in time, the processing of data streams should be completed in real time. Moreover, the process is incremental. Once processed, data are not expected to be retrieved unless a crash occurs. Table 1 is a summary of differences between traditional and stream data processing.

Table 1. Differences between traditional and streaming data processing [1].

	Traditional	**Streaming**
Number of passes	Multiple	Single
Processing time	Unlimited	Restricted
Memory usage	Unlimited	Restricted
Distributed	No	Yes

However, most of the existing research studies tend to construct dynamic ST indexes in real-time batch mode, not in streaming mode. Data in a stream are selected by a window mechanism, shown in Figure 1. In batch mode, the window is driven by the querying requests. One query refers to a window and bulk-loads [2,3] an ST index. The bulk-load algorithm has a better construction efficiency, usually. However, data might not be visible in the index for query once they arrive. Otherwise, two adjacent querying windows may have many duplicated data. If the querying request is too frequent, the resource waste cannot be ignored. In streaming mode, there is only one index. It updates in the meantime while data inflow, even if only one piece of data approaches. The fresh data are inserted while the old data are removed. The process is incremental, passive, and driven by the data themselves. No matter how many requests there are, it can work with low resource consumption. Hence, in the case of online analysis, we would like to obtain an index that performs in streaming mode.

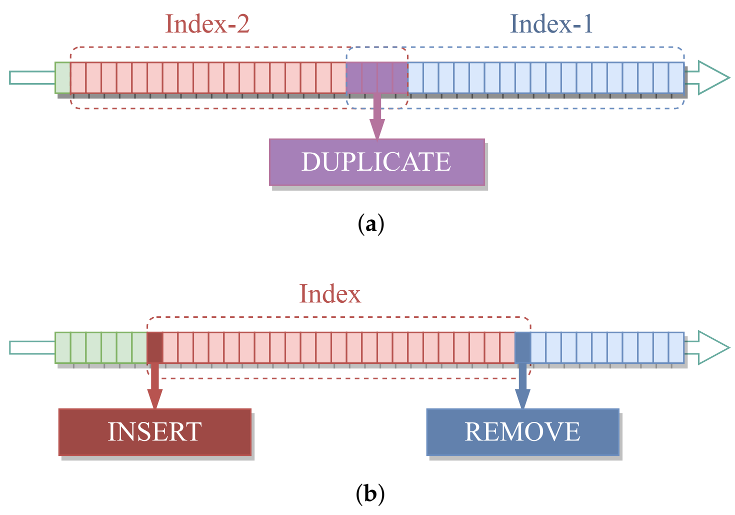


Figure 1. Batch indexing and streaming indexing. (**a**) Batch mode; (**b**) streaming mode.

In this paper, we will discuss how to construct an R-tree index in streaming mode and keep the memory cost and the time cost both under control. Here are our key contributions:

- To restrict the scale of the tree index, we append a first-in–first-out queue for collection to the traditional R-tree and propose an INSERT and batch-REMOVE (I&BR) mechanism. The scale of the R-tree depends on the volume of the collection, which is a customizable variable.
- To reduce the time cost of frequent updates, we propose a time-identified R-tree (TIR) and make a reverse-batch-REMOVE strategy. The improvement benefits from the temporal property of streaming data.
- We describe a distributed system prototype named the time-identified R-tree farm (TIRF) in order to meet practical application requirements.
- We conduct experiments on three diverse datasets, shown in Section 5.1.2. The results offer evidence that a TIR has a controllable memory cost and a low time cost. Compared with PostGIS in PostgreSQL, a TIRF performs many times faster in building an index and range searching.

2. Related Work

2.1. R-Tree-Based Indexes

In the field of spatio-temporal data processing and management, the R-tree is a highly recognized dynamic index. The spatial indexing strategies of many systems, such as Oracle, PostGIS, and GeoSpark, are based on the R-tree [4]. In 1984, Guttman [5] put forward the R-tree, a dynamic index for spatial searching. In a two-dimensional R-tree, data objects with x–y coordination are collected in minimal bounding rectangles (MBRs). The overlay among MBRs is optimized to be possibly small, and the tree structure is kept balanced. Since then, plenty of variants have come up. The R*-tree [6] is one of the most popular designs. It holds much better searching performance than the R-tree but with a slightly higher implementation cost. There are also combinations of R-trees and other types of index, such as the Hilbert R-tree [7], QRB-tree [8], and grid R-tree [9].

Traditional designs of the R-tree cannot handle massive data due to the huge cost of memory. For a stand-alone case, Alsubaiee et al. [10] integrated an R-tree into an LSM-tree-based system by appending a deleted-key B+ tree. LSM-trees batch updates in memory before being written to disk, thus avoiding random writes and improving the throughput of insertion. Shin et al. [11] replaced the deleted-key B+ tree with an updated memo of spatial objects and proposed an LSM RUM-tree. In distributed environments, for instance, Xia et al. [12] proposed the DAPR-tree, which put data access patterns into consideration.

2.2. Indexing Strategies in Streaming Processing

Three types of strategies, grid-based, curve-based, and bulk-load algorithms, are widely used in streaming processing. To better code the spatial relationship between objects, researchers tend to partition space into grids, such as in GeoHash and Google S2. Liu et al. [13] proposed a GeoHash-based index in distributed memory, managing spatial data by GeoHash strings. Fang et al. [14] proposed a histogram index to support trajectory similarity calculation by partitioning free space into disjoint equal-width grid cells. Yang et al. proposed a monitoring mechanism to solve the imbalanced problem of quad-trees for real-time streaming data. Curve-based strategies transform the two-dimensional space into a one-dimensional space-filling curve, such as the Z curve and Hilbert curve. In DITIR, proposed by Cai et al. [15], a data stream is dispatched into chunks by a Z curve and then managed by an in-memory B+ tree. Bulk-load algorithms are constructing strategies for the R-tree performing in batch mode. STR (sort-tile-recursive), proposed by Leutenegger et al. [16], is an algorithm for fast R-tree construction where the data space is sliced by a k-dimensional tile and sorted by coordinates in each direction. Yang et al. [3] combined a hash table and STR to load an R-tree index for moving objects in a data stream. Zhang et al. [2] proposed a bulk-loading algorithm based on the sorting of sampling data in a time window.

However, how to construct a dynamic index in streaming mode is still an open question. It motivates the research presented here, where a workload-controllable dynamic index scheme for streaming processing is provided.

3. Preliminary

We will introduce our data model and our aim in this section.

A general **message** with location could be described as a tuple, $msg = (tuid, coord, data)$. $coord$ consists of latitude and longitude, and $data$ maintains the extra information of this message. $tuid$ represents the reception time of the message without repetition and is a combination of the timestamp and the UID. One message refers to a unique $tuid$. A **stream** is an infinite sequence of messages, shown in Figure 2.

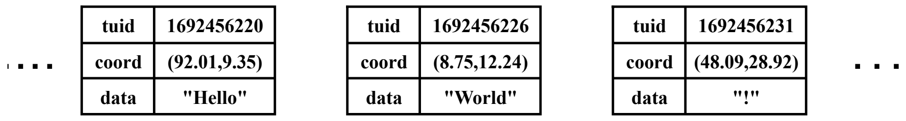


Figure 2. An example of a stream.

We aim to continuously construct an R-tree **index** for fresh data in the stream. We set the variable odt, out-of-date time, to judge whether the message is fresh or not, and set the variable nt to record the $tuid$ of the newest messages. Since the stream is receiving new messages, odt and nt keep increasing and the index updates on and on.

$$S = \{\ldots msg_{i-1}, msg_i, msg_{i+1}, \ldots\}, msg_i.tuid < msg_{i+1}.tuid \tag{1}$$

$$Index = RTree(\{msg_i \mid odt < msg_i.tuid \leq nt, msg_i \in S\}) \tag{2}$$

In addition, we will test the effectiveness of our index by range searching. We give a bounding box, $BB = [< t_{min}, t_{max} >, < lat_{min}, lat_{max} >, < lon_{min}, lon_{max} >]$ and find out all the points within the box by the index.

4. Methodology

In this section, we first apply an INSERT and batch-REMOVE (I&BR) mechanism to a traditional R-tree and propose a time-identified R-tree (TIR). Then we put forward a distributed system prototype, a time-identified R-tree farm (TIRF), for spatio-temporal streaming data processing.

4.1. R-Tree Index

The R-tree [5] is an extension of the B-tree in k dimensions (k = 2, in most cases) with a dynamic balanced height. In a two-dimensional R-tree, data objects with x–y coordination are collected in minimal bounding rectangles (MBRs). As is shown in Figure 3, a node corresponds to a rectangle or a point, and overlap between rectangles is permitted. Child nodes are on the edge of or within the rectangle of their parent nodes. In particular, the number of entries in each node is limited. If an INSERT operation causes overflow (number of entries exceeds M), then the node is split into two nodes. If a REMOVE operation leads to underflow (number of entries is less than m, $m \leq \frac{M}{2}$), then the node is removed and all its child nodes are reinserted into the tree. This is the main mechanism for how the tree structure remains balanced.

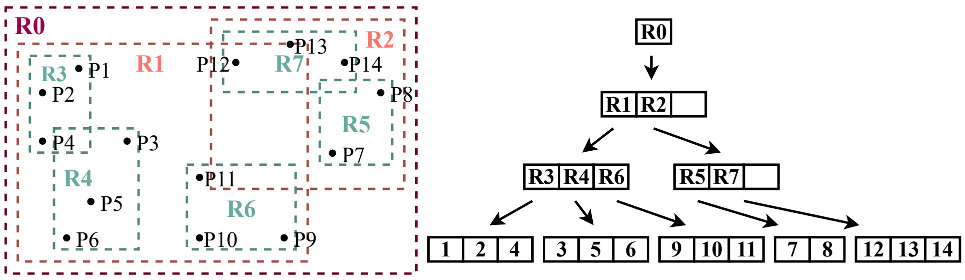
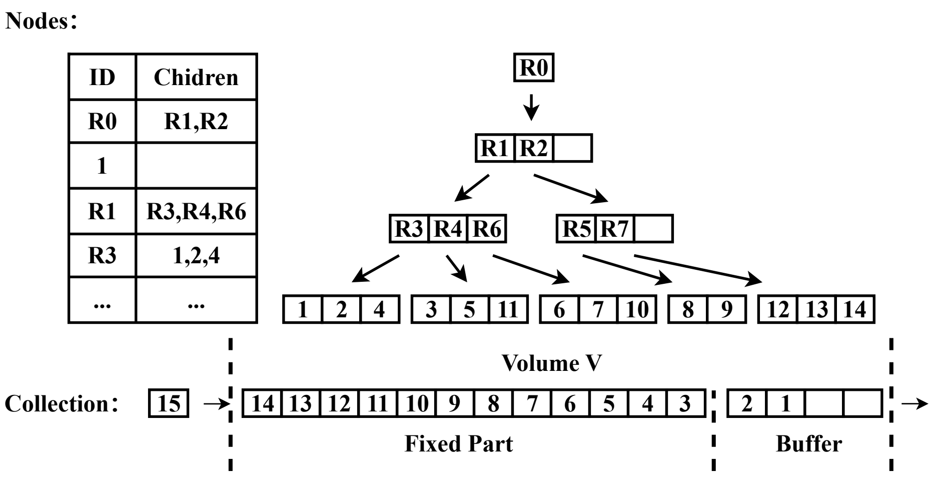


Figure 3. An example of an R-tree.

4.2. *R-Tree with I &BR*

A traditional tree index grows while data are inserted. An R-tree and its variants go through a performance reduction when the height of the tree grows. The higher it is, the more layers a top-to-bottom operation has to pass through. In addition, when the memory runs out, partial nodes are saved on disk. Updates of nodes require frequent access to the disk, which produces extra time costs. If the scale of the tree is restricted, the performance of the R-tree is guaranteed. Hence, we come up with the idea of limiting the scale of the R-tree. Figure 4 shows our design.



Nodes:

ID	Chidren
R0	R1,R2
1	
R1	R3,R4,R6
R3	1,2,4
...	...

Figure 4. R-tree with I&BR.

Along with the tree index, we append a message collection. **Collection** collects the message from the stream (inserted as it inflows). It is a first-in–first-out queue and performs as a sliding window. Every message in the queue is related to a leaf node in the R-tree. In the data stream, we cannot predict the arrival time of the next message, so we insert messages one by one. The fresher messages queue behind the older ones. Messages in the **buffer** are the ones about to be popped. The collection has a fixed volume, V. Then the maximum buffer can be set as $p \times V$, and p is a percentage. Once the buffer is filled, all messages in the buffer will be popped and the relative leaf nodes will be removed. In this way, the height of the R-tree is limited to $logV$. The process is what we call I&BR. INSERT and REMOVE (I&R) can be implemented while the maximum buffer is set as 1. In stream processing, batch updating is a practical, feasible method [14]. It is also verified in Section 5.2.

An R-tree with a controllable scale can be entirely reserved in memory, but the historical information is lost when batch REMOVE is performed. Then, persistent storage is requested. Every time we have pushed a certain amount of messages into the collection, a serialization for the R-tree is executed. The amount could be the collection volume, V, if we try to store data without repeat.

An adjacent linked list is used for maintaining nodes in the R-tree. The pointer between the parent node and the child node is kept by node ID. Then, in serialization or deserialization of the tree index, we can just iterate over the list instead of using a breadth-first search in the memory. It saves time in I/O. Moreover, since the R-tree is a high-balanced tree index, with the height of the tree being fixed, the list's capacity can be predefined. It is much more convenient for memory allocators to arrange the usage of resources.

The process of I&BR is shown in Algorithm 1.

Algorithm 1 INSERT and batch REMOVE (I&BR)

```
    Input: $msg$
 1: $rtree.insert(msg)$
 2: $collection.push(msg)$
 3: if have pushed enough $msg$ then
 4:     $rtree.serialize()$
 5: end if
 6: if Buffer is filled then
 7:     while Buffer is not empty do
 8:         $msg \leftarrow buffer.pop()$
 9:         $rtree.remove(msg)$
10:     end while
11: end if
```

4.3. Time-Identified R-Tree

Having restricted the memory, we discuss the method to accelerate the I&BR process in this section.

There are two ways to reduce building time. One is bulk loading [16], which requests batch INSERT. As mentioned previously, the arrival time of messages in a stream is not predictable, so we do not tend to use batch INSERT. The other is using trees supporting frequent updates, such as RUM [17] and FUR [18], which are object-intensive. FUR applies a bottom-up algorithm instead of a top-down strategy for updating the location of moving objects in a traditional R-tree. RUM appends an update memo that keeps the latest entries of numbered objects. As our method is data-intensive, we can learn from RUM and FUR. Our core idea is to build direct access to leaf nodes and remove nodes by the properties of the stream.

For every message, we give a *tuid* to identify it. Then, we record the node ID of the related leaf node in the message saved in the collection and change tuples in the collection to $msg = (tuid, coord, data, nodeID)$. Hence, the leaf node is directly accessible, obtaining the streaming and temporal properties of the collection. It could be said that the R-tree is identified by *tuid*. We name it a time-identified R-tree (TIR).

We reformulate the batch-REMOVE strategy in the TIR. Figure 5 is an example of batch REMOVE. Since the collection is a first-in–first-out queue, the general idea is to remove items in order. However, affected by the INSERT order, leaf nodes identified by smaller *tuid*s are in front of the greater and are accessed before them in a REMOVE operation. Once the target of REMOVE has been accessed, the operation breaks. To remove the greater leaf nodes, we have to execute the same top-to-bottom search. It is a waste of time. However, if we reverse the REMOVE order, it will be different. We can find nodes that are out-of-date in advance. For example, while searching for node 7, we can in the meantime find node 6 and remove it. When it is up to the REMOVE for node 6, we just need to directly access it by nodeID and check whether it has been removed.

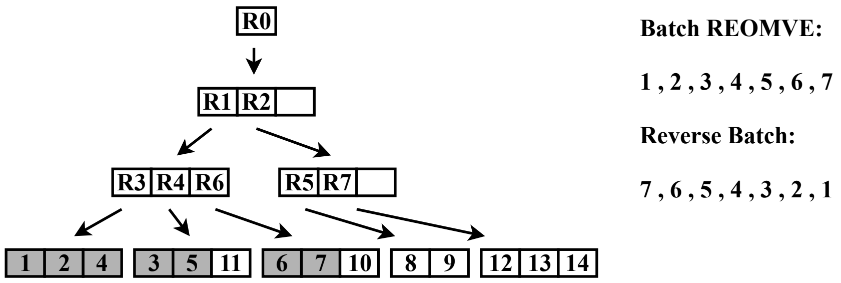


Figure 5. An example of batch REMOVE.

We assume that leaf nodes with the same parent are strictly ordered by *tuid*. We set n as the number of batch REMOVE, N as the number of leaf nodes in the tree and E as the average number of out-of-date leaf nodes in a parent node. With the reverse batch-REMOVE strategy, the time complexity is $O(\frac{n}{E} \cdot \log N + n)$, while with the general strategy, $O(n \cdot \log N)$. The I&BR process of the TIR is shown in Algorithm 2. In addition, the complexity of $parent.remove(leaf)$ in line 16 is based on the data structure storing child nodes. If the structure is a list, then it is $O(N)$ at the worst. If the structure is a deque, then the best situation is $O(1)$ but requires the strict order of child nodes, which may introduce extra cost. In our implementation, we select a list as the structure.

Algorithm 2 I&BR in TIR

```
    Params: $odt$
    Input: $msg$
 1: $rtree.insert(msg)$
 2: $collection.push(msg)$
 3: if have pushed enough $msg$ then
 4:     $rtree.serialize()$
 5: end if
 6: if Buffer is filled then
 7:     $buffer.reverse()$
 8:     while Buffer is not empty do
 9:         $msg \leftarrow buffer.pop()$
10:         $n \leftarrow msg.nodeID$
11:         if $n$ has not been removed then
12:             Begin REMOVE $n$
13:             ...
14:             for $leaf$ in $parent.children$ do
15:                 if $leaf$ is $n$ or $leaf.tuid \leq odt$ then
16:                     $parent.remove(leaf)$
17:                 end if
18:             end for
19:             ...
20:             End REMOVE $n$
21:         end if
22:     end while
23: end if
```

Two cases may break the order between leaf nodes, SPLIT in the INSERT operation and REINSERT in the REMOVE operation. In SPLIT, we reorder the nodes if the relative sequence is changed, shown in Algorithm 3. In REINSERT, we filter out the out-of-date nodes first, and for the others, we insert them into the children lists instead of pushing into them, as shown in Algorithm 4.

Algorithm 3 SPLIT in TIR

```
1: $parentNode \leftarrow node.parent$
2: $leftNode, rightNode \leftarrow SPLIT(node)$
3: **if** relative orders in $leftNode, rightNode$ have been changed **then**
4:     $leftNode.sortByTuid()$
5:     $rightNode.sortByTuid()$
6: **end if**
7: $parentNode.remove(node)$
8: $parentNode.insert(leftNode, rightNode)$
```

Algorithm 4 REINSERT in TIR

```
    **Param:** $odt$
    **Input:** $msg$
1: **if** $msg.tuid \leq odt$ **then**
2:     END REINSERT
3: **end if**
4: $pn \leftarrow chooseParentNode(msg)$
5: **for** $c_i$ in $pn.children$ **do**
6:     **if** $c_i.tuid < msg.tuid$ and $c_{i+1}.tuid \geq msg.tuid$ **then**
7:         $pn.children.insert(msg, i+1)$
8:     **end if**
9: **end for**
10: **if** $msg$ is not inserted **then**
11:     $pn.children.push(msg)$
12: **end if**
```

4.4. Time-Identified R-Tree Farm

In many streaming processing tasks, distributed computing and storage are required. To meet practical application requirements, we propose a time-identified R-tree farm, a distributed system prototype for spatio-temporal streaming data processing. Our architecture is shown in Figure 6. It consists of a **stream**, a **diverter**, **R-trees**, a **warehouse**, and **gates**.

A **stream** integrates messages from different sources. As is mentioned in Section 3, messages in a stream are kept in order by *tuid*. A **Diverter** selects and divides the stream. The dividing strategy decides the set and the arrival time of received messages in every tree and then influences the growth of the tree. On the one hand, if we expect to ensure the throughput, we can divide the stream based on the consumption rates of trees. The faster the tree consumes messages, the more messages it will receive. On the other, if we assign different regions to each tree, only messages located in the region will be sent to that tree.

In a distributed system, a worker could maintain an in-memory R-tree or TIR in our implementation. **R-trees** keep updating while data inflow. They take in new messages and drop off the old ones. Through **Gate-1**, we can monitor every change in the R-trees. Collection in the TIR roles as an adaptive time window, while the TIR itself is an incremental result of indexing.

Once in a while, snapshots of R-trees are made and then serialized in a storable form (in our implementation, the ".json" file format). We call the fixed R-tree wood. The **warehouse** stores the woods and manages them. The warehouse could be a key-value-based distributed database or just a distributed file system. It is a permanent storage on disk. Through **Gate-2**, we can query historical data and reuse the indexes created before.

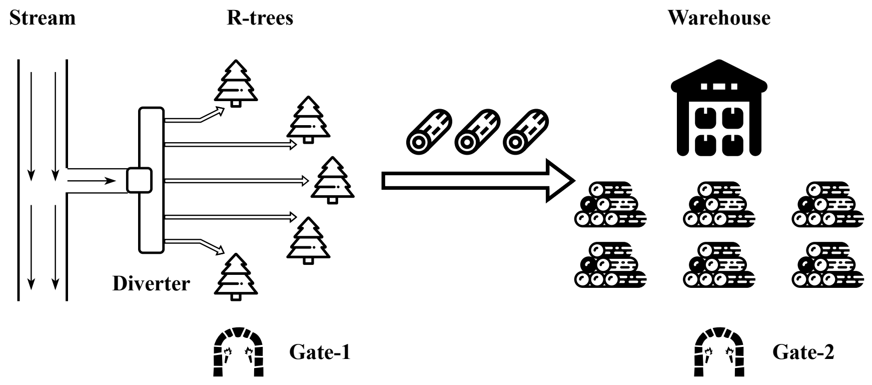


Figure 6. The architecture of the TIRF.

5. Experimental Evaluation

5.1. Experiment Setup

5.1.1. Environment

All experiments are implemented on an SMP (symmetric multiprocessor) server with 16 CPU cores at 4.50 GHz. The RAM size is 64 GB.

5.1.2. Dataset

We use three datasets with different spatial distributions, as shown in Figure 7. Table 2 shows the details. They are as follows:

(1) **Random**: the dataset is a series of random points spread all over the world evenly, generated by computer. (2) **ADSB**: The dataset (https://www.adsbexchange.com/, accessed on 12 October 2023) includes trajectories of flights in one day. The distribution is meshlike and centered on several locations. (3) **AIS**: This dataset (https://marinecadastre.gov/ais/, accessed on 12 October 2023) includes trajectories of vessels during a month period. Records are concentrated on the shore and the ports. Since ADSB and AIS are datasets from the real world, we can better evaluate the performance in practice. Each dataset is sorted by *tuids* of messages and was stored in files previously. In addition, to test the throughput, the data will be consumed as fast as the workers are able to consume them.

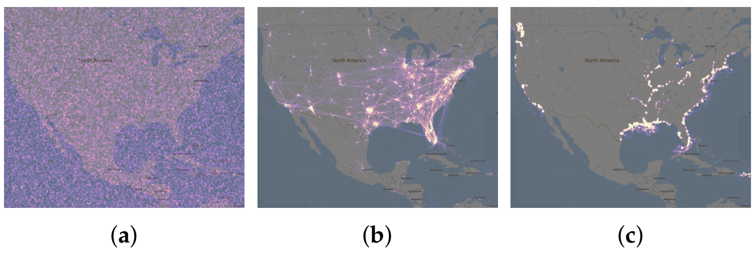


Figure 7. Distributions of datasets. (**a**) Random; (**b**) ADSB; (**c**) AIS.

Table 2. Details of datasets.

	Random	ADSB	AIS
Data type	Points	Points	Points
Sample size	100 million	100 million	100 million
Sample frequency	-	5 s	1 to 90 s
Time range	-	1 day	23 days
Spatial distribution	Even	Meshlike	Dotted

5.1.3. Parameters and Explanation

(1) **R-tree**: We set the maximum of children in a node as 32 and the minimum as 6. In a full R-tree with a height equaling 3, the amount of leaf nodes is 32,768 (32^3), and with a height of 4, the number is 1,048,576 (32^4).

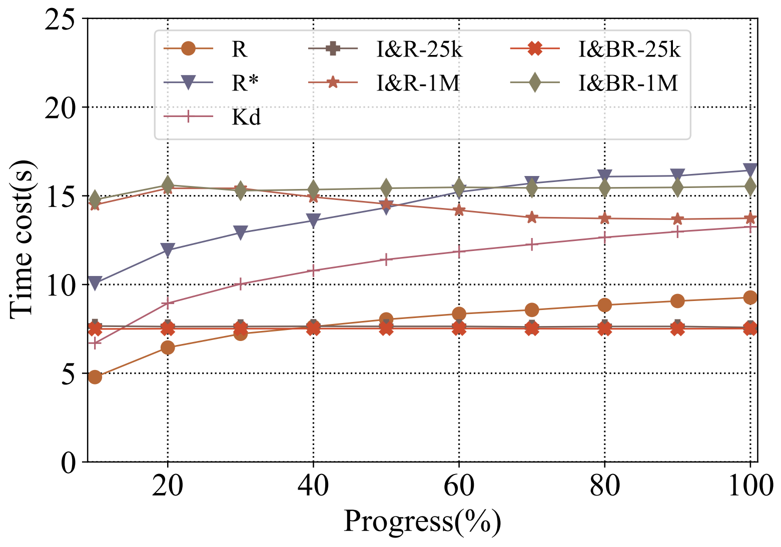
(2) **Collection**: The collection mechanism needs extra parameters. One is the volume of the collection, V. For instance, $V = 25k$ means the maximum number of messages in the collection is 25,000; $V = 1M$ means 1 million. The other is the percentage of buffer in the collection, p. With $V = 1M, p = 1\%$, the buffer size is 10,000.

5.2. Cost of Construction

To evaluate the performance, we conduct experiments to test the time cost and the memory cost during the construction process. We test an R-tree, R*-tree, Kd-tree (a K-dimensional tree for 2D attributes), R-tree with I&R, R-tree with I&BR, and time-identified R-tree (TIR). All the codes are implemented in RUST language (source code will be pushed to Gitee https://gitee.com/Vill-V-V/time-identified-rtree, accessed on 3 January 2024). This set of experiments is executed purely in memory, with little I/O. The index serialization operation in the R-tree with I&R, R-tree with I&BR, and TIR is executed but does not save the serialized index to disk.

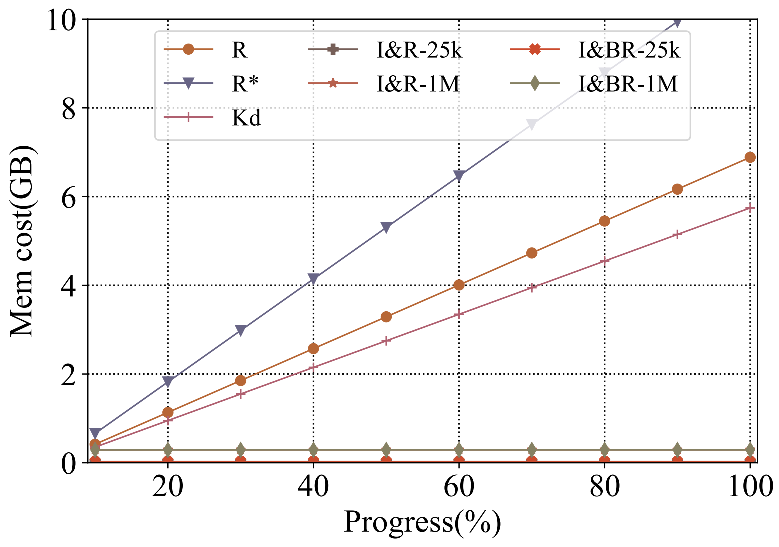
In the first place, we test the cost of a traditional R-tree, Kd-tree, and R-tree with I&BR. We continuously input points in the Random dataset and record the time cost and the memory cost in progress. The height of a traditional R-tree is unlimited, while I&BR restricts the height by V. To compare the effect of different tree heights, we set the volume as $25k$ (height 3) and $1M$ (height 4). To evaluate the batch-remove strategy without any other improvement, we test I&R ($p \times V = 1$) and I&BR ($p = 10\%$).

The results are shown in Figure 8. The x axis is the progress of the construction process. For example, when the progress approaches 10%, 10 million messages have been consumed. The y axis is the total time cost in every 10% of the process in Figure 8a and the average memory cost at every 10% of the process in Figure 8b. With the growth of the tree, the time costs of the R-tree , Kd-tree, and R*-tree show a logarithmic increase. Regarding the aspect of memory cost, traditional R-trees occupy more and more resources while data inflow in a linear growth trend. Trees with $V = 1M$ need more time and more memory than those with $V = 25M$, but both of them keep the costs under control.



(**a**)

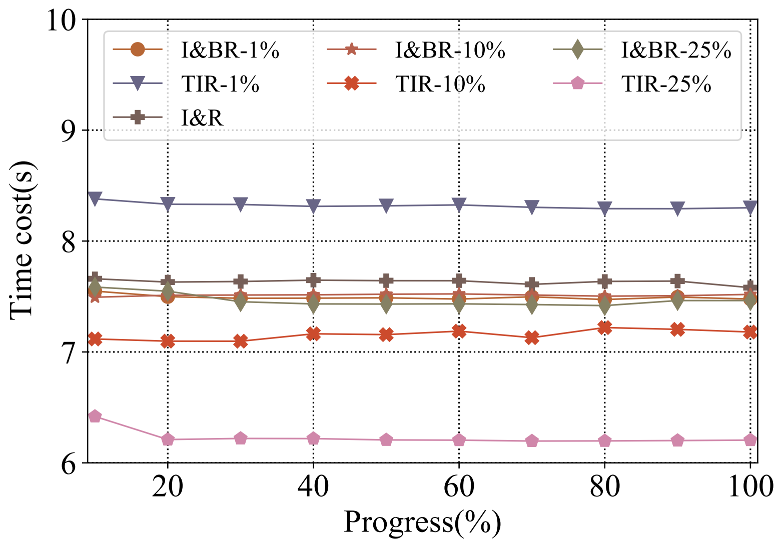
Figure 8. *Cont.*



(**b**)

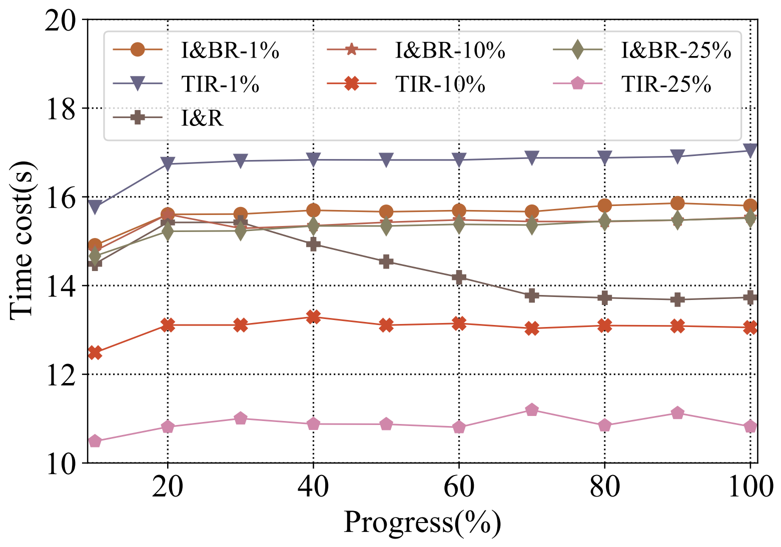
Figure 8. Comparison with traditional R-tree. (**a**) Time cost; (**b**) Memory cost.

In order to obtain a better contrast of trees with different sizes of buffer and evaluate the effectiveness of the changes made in the TIR, we perform two sets of experiments, shown in Figure 9. Firstly, we focus on I&R and I&BR with different p. In the case of $V = 25k$, it shows that a tree with a greater buffer size may cost less time, and in the case of $V = 1M$, the advantage of batch removal appears to be distinct. TIR-1% performs a little worse than the R-tree with I&BR because the calculation is more complex. However, as the size of the buffer in the TIR increases, there is an obvious cost reduction. It shows that the TIR with $p = 25\%$ performs much better than the others. In our scenario, we expect that the in-memory index can serve for online queries. The collection has to keep enough messages for query, so we stop our experiment at $p = 25\%$.



(**a**)

Figure 9. *Cont.*



(**b**)

Figure 9. Comparison among I&R, I&BR, TIR. (**a**) $V = 25k$; (**b**) $V = 1M$.

We repeat the above experiments with the other two datasets. The results of time cost are shown in Table 3. In addition, the memory cost for trees with $V = 25k$ is about 29 MB, and for those with $V = 1M$, it is about 296 MB. We use I&R as the baseline and calculate the average reduction rate. I&BR shows little improvement in time cost. Furthermore, with a small buffer, the changes in the TIR even have negative effects. The TIR with a big buffer reduces time costs to a large extent and reaches a rate of 1 million points per second in some cases.

The experiments show that the R-tree with I&BR can restrict its scale and height by V and p, making memory cost and time cost of construction predictable and controllable. TIR accelerates the building process, making the latency even less.

Table 3. Total time cost(s).

	Random	**ADSB**	**AIS**	**(%)**
R	78.21	55.06	58.86	-
R*	142.43	115.39	107.04	-
Kd	110.88	105.26	108.17	-
IR-25k	76.32	77.24	79.86	Baseline
IBR-25k-1%	74.92	78.22	82.12	−0.79
IBR-25k-10%	75.12	79.79	83.32	−2.06
IBR-25k-25%	74.66	79.59	82.46	−1.41
TIR-25k-1%	83.19	82.53	92.72	−10.72
TIR-25k-10%	71.55	73.06	79.13	4.15
TIR-25k-25%	**62.28**	**67.70**	**69.89**	**14.37**
IR-1M	143.91	161.99	199.36	Baseline
IBR-1M-1%	156.30	163.64	200.11	−2.93
IBR-1M-10%	153.83	164.27	198.42	−2.23
IBR-1M-25%	153.00	159.72	193.09	−0.11
TIR-1M-1%	167.52	159.62	208.13	−5.94
TIR-1M-10%	130.53	111.93	138.61	24.58
TIR-1M-25%	**108.86**	**105.43**	**120.49**	**33.74**

5.3. Simple Implementation of TIRF

We have evaluated the construction performance of a TIR in a single worker. Does a TIR work in distributed or parallel computing tasks? How is the query performance of a

TIR? To verify the availability of a TIR in multiworkers, we implement a simple TIRF in a parallel environment.

For the process of constructing the index, we set one loading worker and N indexing workers. The loading worker's role is of the stream and the diverter in TIRF. It loads the original data from the file system, generates a data stream, and sends the messages to the indexing workers. The rule of the diverter is "the unoccupied worker gets the message". Each indexing worker maintains a TIR, constructs the index individually, and writes the serialized index into the file system concurrently. The file system's role is of the warehouse in the TIRF.

For the process of range searching, we set N querying workers. Each worker loads the indexes it constructed before from the file system and answers the query requests one by one.

5.4. Performance of the TIRF

We design this set of experiments to evaluate the practicability of the TIRF. We choose PostgreSQL, which is a widely used relational database, as the comparison. In this set of experiments, the performance is influenced not only by calculation in memory but also by I/O with disk.

We test the total time of building the index for 100 million points first. In PostgreSQL, we use the spatial extension PostGIS and we use GiST as the index and force the parallel mode on. We test two types of indexes. One is GiST by only geolocation (GiST-2D), and the other is GiST by *tuid* and geolocation (GiST-3D). The points data are preloaded into a table in the database. We record the response time of the "CREATE INDEX" instruction. In the TIRF, we set 1 loading worker and 15 indexing workers and record the time of loading data files for the loading worker and the total time of building the index, serializing to files, and saving to disk for the indexing workers. Two cases are considered, a TIRF with $V = 25k$ (TIRF-25k) and a TIRF with $V = 1M$ (TIRF-1M). Table 4 shows the time cost of the index-building process.

Table 4. Building time(s).

	Random	**ADSB**	**AIS**
GiST-2D	475.65	414.04	440.25
GiST-3D	12,243.74	13,378.72	14,153.62
TIRF-25k	85.96	71.87	71.28
TIRF-1M	101.86	82.87	84.25

Then we test the search performance. There are two types of time ranges, 1% (R = 1%) and 10% (R = 10%), standing for two querying situations. The target of querying is the amount of points in the bounding box. For each situation, we generate 100 random bounding boxes and continuously execute these range-searching queries. In PostgreSQL, we conduct two rounds for each set of queries. The database has to buffer the index and the data into memory in the first round, while in the second round, the process is completely in memory with little I/O cost. By subtracting the time costs of two rounds, we can estimate the time cost of I/O in PostgreSQL. In the TIRF, we also use 15 querying workers for searching, and we record the time costs. We divide the process into two parts. One is loading indexes and data into memory (I/O time cost), and the other is searching and calculating the query results with all the necessary information loaded into memory before (CPU time cost). Tables 5 and 6 show the results of the experiments.

Table 5. Loading time(s).

	Random		**ADSB**		**AIS**	
	1%	**10%**	**1%**	**10%**	**1%**	**10%**
GiST-2D	9.47	9.35	9.08	9.71	9.61	9.75
GiST-3D	111.09	166.98	70.96	168.74	66.91	164.48
TIRF-25k	13.92	11.81	12.38	13.67	11.24	11.06
TIRF-1M	16.76	14.17	14.94	17.56	13.39	14.73

Table 6. Searching time(s).

	Random		**ADSB**		**AIS**	
	1%	**10%**	**1%**	**10%**	**1%**	**10%**
GiST-2D	60.56	67.98	61.98	68.96	58.76	66.55
GiST-3D	24.16	231.35	13.29	129.63	17.24	139.63
TIRF-25k	0.11	0.08	0.14	0.66	0.56	0.90
TIRF-1M	1.05	0.79	3.45	2.16	2.09	5.28

In PostgreSQL, GiST-2D is an index with two dimensions of attributes and GiST-3D is with three dimensions of attributes. The building cost of GiST-3D is much greater than GiST-2D. However, GiST-3D spends more time in I/O than GiST-2D and has worse performance in a large time range of querying. The reason is that in such a complex index as GiST-3D, only one core can be used to select the answers, while GiST-2D can roughly filter data that satisfy the spatial condition and use multiple cores to judge which message fits the temporal condition. The amount of judgments in GiST-2D is massive and not changed by the time range. Because of the lack of temporal information, range-querying tasks with time intervals in GiST-2D are CPU-bounded. In addition, the searching performance of GiST-3D in range-1% is great. This shows that the time attribute benefits tasks with small time intervals.

As for the TIRF, it constructs indexes at a high speed and selects data for range querying efficiently. The cost of loading indexes is in the majority of total cost. It confirms that the TIRF is I/O bounded.

The experiments show that a TIR in architectures like a TIRF works better than GiST-2D and GiST-3D. A TIR is constructed only based on the spatial attribute, just as GiST-2D is, retaining the performance in index loading and spatial querying. Due to the I&BR mechanism and continuous serialization, the TIR maintains the temporal attribute and builds the index piece by piece. This makes time-range judgments in a TIR much easier than in GiST-2D.

5.5. Evaluation of Parallelism

We have seen that a TIR is effective in a multiworker situation in former experiments. However, whether the parallel environment improves the performance is still a question. In this section, we conduct experiments with different numbers of workers and evaluate the parallelism.

We conduct the construction task with 1 loading worker and 1, 3, 6, 9, 12, and 15 indexing workers and record the average indexing time and the loading time. Then we record the average searching time and the average loading time among querying workers (the number is the same as indexing workers) in range-querying tasks. The results of experiments on the random dataset are shown in Figures 10 and 11. More detailed statistics can be found in Appendix A.

Figure 10 shows the proportion of loading time and indexing time in constructing the index. The loading time is relatively stable because this part of the task is completed by only a loading worker. The indexing time is reduced while the number of workers increases but approaches a limitation in the end. Since the indexing part contains writing the serialized index to disk, we suppose the limitation is the I/O bound.

Figure 11 shows the proportion of searching in loaded indexes is much smaller than loading the indexes as well. Compare $V = 25k$ with $V = 1M$. Few querying workers can reach the I/O bound when the index size is small. A parallel strategy helps in loading when the index size is large. However, the searching time is not obviously reduced by multiworkers. The stream is divided evenly by the diverter. It may separate messages with close locations to different workers and then increase the complexity of range querying.

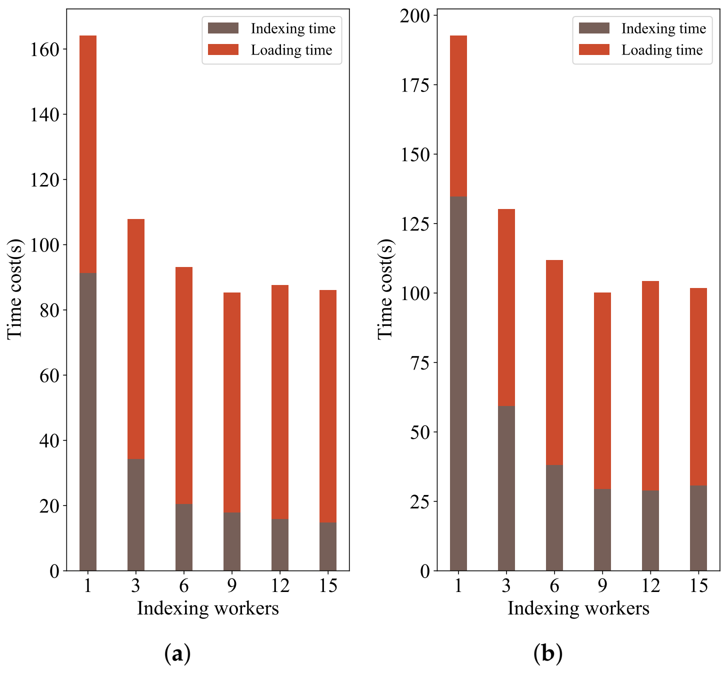


Figure 10. Constructing index with different numbers of workers. (**a**) $V = 25k$; (**b**) $V = 1M$.

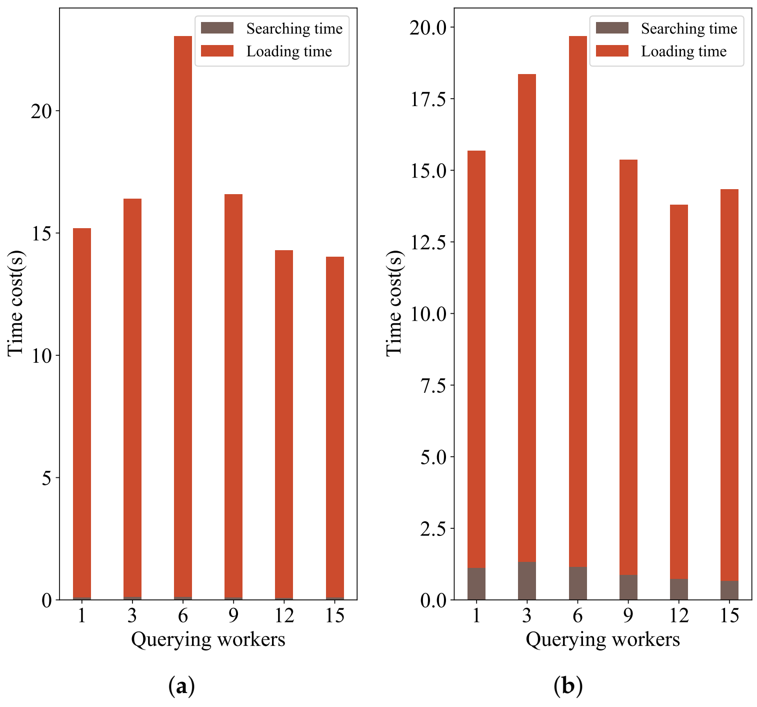
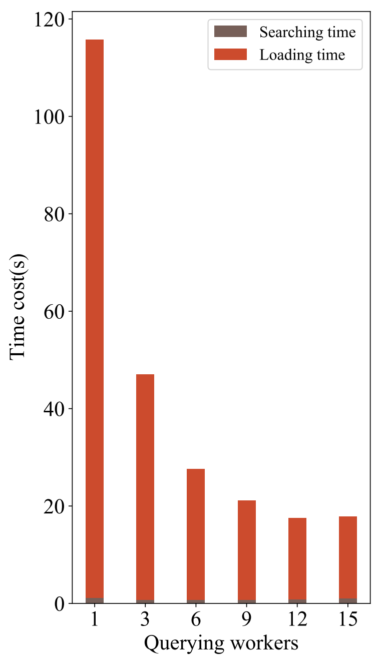
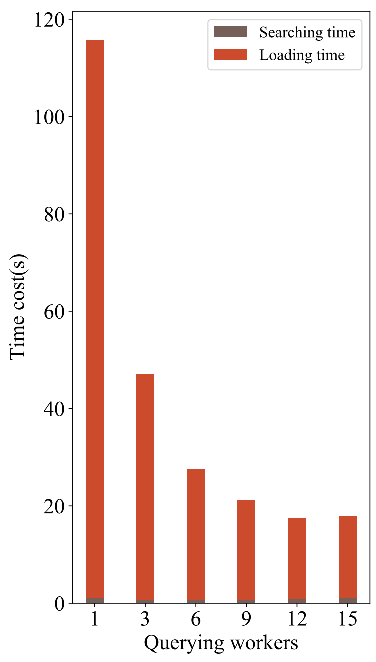


Figure 11. *Cont.*



(**c**)



(**d**)

Figure 11. Range querying with different numbers of workers. (**a**) $V = 25k$, R = 1%; (**b**) $V = 25k$, R = 10%; (**c**) $V = 1M$, R = 1%; (**d**) $V = 1M$, R = 10%.

It is confirmed that by benefiting from stable memory usage and low time cost in updating, a TIR can take the responsibility of a spatio-temporal index in distributed or parallel tasks. The bottleneck is the I/O bandwidth in a TIRF. How to increase the speed of data input and index output is the key to further improve the throughput. To improve the query performance in a multiworker situation, a spatio-temporal-based dispatching strategy for the diverter and an index management mechanism in the warehouse are required. They are the points to be discussed in future work.

6. Conclusions

We propose a time-identified R-tree, a dynamic spatio-temporal index for streaming processing, which has controllable memory cost and low time cost. To control the scale, we append a message collection to a traditional R-tree and formulate an INSERT and batch-REMOVE mechanism. To improve throughput, we propose a reverse-batch-REMOVE strategy. Experiments confirm that the time-identified R-tree performs better than the traditional R-tree when constructing an index for a data stream. Moreover, we describe a distributed system prototype for a spatio-temporal data stream, the time-identified R-tree farm. It is a novel but incomplete design, and experiments show that it has potential.

Author Contributions: Conceptualization, Weichen Peng, Luo Chen, Xue Ouyang and Wei Xiong; methodology, Weichen Peng; formal analysis, Weichen Peng; resources, Luo Chen; software, Weichen Peng; writing—original draft preparation, Weichen Peng; writing—review & editing, Xue Ouyang and Wei Xiong; project administration, Luo Chen; funding acquisition, Luo Chen. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (grant no. 41971362).

Data Availability Statement: Publicly available datasets were analyzed in this study. And the supplementary data can be found in https://www.adsbexchange.com/ and https://marinecadastre.gov/ais/, accessed on 12 October 2023.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

Table A1. Constructing an index with different numbers of workers.

	Random		**ADSB**		**AIS**	
	Load(s)	**Index(s)**	**Load(s)**	**Index(s)**	**Load(s)**	**Index(s)**
1-TIRF-25k	72.75	91.35	57.40	90.87	58.41	98.84
3-TIRF-25k	73.44	34.38	44.88	36.00	58.79	36.22
6-TIRF-25k	72.64	20.47	57.45	21.53	58.22	21.90
9-TIRF-25k	67.43	17.82	59.67	17.84	56.62	19.65
12-TIRF-25k	71.72	15.92	56.75	15.96	53.72	16.02
15-TIRF-25k	71.15	14.81	56.87	15.00	56.28	15.01
1-TIRF-1M	57.81	134.84	59.56	128.68	54.54	147.89
3-TIRF-1M	70.76	59.41	59.05	52.60	56.50	58.51
6-TIRF-1M	73.79	38.05	61.49	34.60	55.46	35.39
9-TIRF-1M	70.65	29.52	59.37	29.10	55.87	31.34
12-TIRF-1M	75.35	28.88	55.27	25.41	57.55	27.53
15-TIRF-1M	71.09	30.77	58.48	24.38	58.56	25.68

Table A2. Range (R = 1%) querying with different numbers of workers.

	Random		**ADSB**		**AIS**	
	Load(s)	**Query(s)**	**Load(s)**	**Query(s)**	**Load(s)**	**Query(s)**
1-TIRF-25k	15.08	0.11	13.44	0.06	13.31	0.24
3-TIRF-25k	16.28	0.12	14.08	0.18	14.04	0.20
6-TIRF-25k	22.91	0.13	19.98	0.10	19.01	0.18
9-TIRF-25k	16.48	0.11	14.82	0.08	13.93	0.17
12-TIRF-25k	14.20	0.09	13.02	0.08	13.36	0.16
15-TIRF-25k	13.92	0.11	11.81	0.08	12.38	0.14
1-TIRF-1M	114.65	1.09	122.47	1.09	118.17	2.41
3-TIRF-1M	46.24	0.74	43.25	0.70	43.82	1.58
6-TIRF-1M	26.86	0.73	22.06	0.63	22.02	1.24
9-TIRF-1M	20.37	0.77	16.68	0.69	17.52	2.26
12-TIRF-1M	16.65	0.84	14.24	0.77	14.62	2.11
15-TIRF-1M	16.76	1.05	14.17	0.79	14.94	3.45

Table A3. Range (R = 10%) querying with different numbers of workers.

	Random		**ADSB**		**AIS**	
	Load(s)	**Query(s)**	**Load(s)**	**Query(s)**	**Load(s)**	**Query(s)**
1-TIRF-25k	14.56	1.13	13.68	0.87	13.56	1.00
3-TIRF-25k	17.02	1.32	16.93	0.83	15.81	1.37
6-TIRF-25k	18.53	1.16	15.98	1.05	15.15	1.60
9-TIRF-25k	14.48	0.88	12.72	0.76	13.18	1.15
12-TIRF-25k	13.05	0.74	11.81	0.72	10.84	1.01
15-TIRF-25k	13.67	0.66	11.24	0.56	11.06	0.90
1-TIRF-1M	95.13	7.59	101.31	7.21	97.78	12.33
3-TIRF-1M	38.45	3.36	37.26	3.18	33.73	5.40
6-TIRF-1M	28.91	2.46	20.78	2.11	17.65	3.40
9-TIRF-1M	19.68	1.96	14.91	1.85	15.84	3.45
12-TIRF-1M	17.47	2.01	13.86	1.76	12.89	3.52
15-TIRF-1M	17.56	2.16	13.39	2.09	14.73	5.28

References

1. Gama, J.; Gaber, M.M. (Eds.) *Learning from Data Streams*; Springer: Berlin/ Heidelberg, Germany, 2007. [CrossRef]
2. Zhang, T.; Yang, L.; Shen, D.; Fan, Y. An Efficient In-Memory R-Tree Construction Scheme for Spatio-Temporal Data Stream. In *Proceedings of the Service-Oriented Computing–ICSOC 2018 Workshops*; Liu, X., Mrissa, M., Zhang, L., Benslimane, D., Ghose, A., Wang, Z., Bucchiarone, A., Zhang, W., Zou, Y., Yu, Q., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2019; pp. 253–265. [CrossRef]
3. Yang, L.H.; Shen, D.H.; Fan, Y.L.; Gao, N. A Moving Object Spatial Index for Spatio-Temporal Data Stream. *Acta Electonica Sin.* **2021**, *49*, 992. [CrossRef]
4. Wang, S.; Bao, Z.; Culpepper, J.S.; Cong, G. A Survey on Trajectory Data Management, Analytics, and Learning. *ACM Comput. Surv.* **2021**, *54*, 39:1–39:36. [CrossRef]
5. Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD Rec.* **1984**, *14*, 47–57. [CrossRef]
6. Beckmann, N.; Kriegel, H.P.; Schneider, R.; Seeger, B. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD'90, New York, NY, USA, 23–26 May 1990; pp. 322–331. [CrossRef]
7. Kamel, I.; Faloutsos, C. Hilbert R-tree: An Improved R-tree Using Fractals. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB'94, San Francisco, CA, USA, 12–15 September 1994; pp. 500–509.
8. Yu, J.; Wei, Y.; Chu, Q.; Wu, L. QRB-tree Indexing: Optimized Spatial Index Expanding upon the QR-tree Index. *ISPRS Int. J. Geo-Inf.* **2021**, *10*, 727. [CrossRef]
9. Goyal, P.; Challa, J.S.; Kumar, D.; Bhat, A.; Balasubramaniam, S.; Goyal, N. Grid-R-tree: A Data Structure for Efficient Neighborhood and Nearest Neighbor Queries in Data Mining. *Int. J. Data Sci. Anal.* **2020**, *10*, 25–47. [CrossRef]
10. Alsubaiee, S.; Behm, A.; Borkar, V.; Heilbron, Z.; Kim, Y.S.; Carey, M.J.; Dreseler, M.; Li, C. Storage Management in AsterixDB. *Proc. VLDB Endow.* **2014**, *7*, 841–852. [CrossRef]
11. Shin, J.; Wang, J.; Aref, W.G. The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 2285–2290. [CrossRef]
12. Xia, J.; Huang, S.; Zhang, S.; Li, X.; Lyu, J.; Xiu, W.; Tu, W. DAPR-tree: A Distributed Spatial Data Indexing Scheme with Data Access Patterns to Support Digital Earth Initiatives. *Int. J. Digit. Earth* **2020**, *13*, 1656–1671. [CrossRef]
13. Liu, J.; Li, H.; Gao, Y.; Yu, H.; Jiang, D. A Geohash-Based Index for Spatial Data Management in Distributed Memory. In Proceedings of the 2014 22nd International Conference on Geoinformatics, Kaohsiung, Taiwan, 25–27 June 2014; pp. 1–4. [CrossRef]
14. Fang, Z.; Gong, S.; Chen, L.; Xu, J.; Gao, Y.; Jensen, C.S. Ghost: A General Framework for High-Performance Online Similarity Queries over Distributed Trajectory Streams. *Proc. ACM Manag. Data* **2023**, *1*, 173. [CrossRef]
15. Cai, R.; Lu, Z.; Wang, L.; Zhang, Z.; Fur, T.Z.J.; Winslett, M. DITIR: Distributed Index for High Throughput Trajectory Insertion and Real-Time Temporal Range Query. *Proc. VLDB Endow.* **2017**, *10*, 1865–1868. [CrossRef]
16. Leutenegger, S.; Lopez, M.; Edgington, J. STR: A Simple and Efficient Algorithm for R-tree Packing. In Proceedings of the 13th International Conference on Data Engineering, Birmingham, UK, 7–11 April 1997; pp. 497–506. [CrossRef]
17. Silva, Y.N.; Xiong, X.; Aref, W.G. The RUM-tree: Supporting Frequent Updates in R-trees Using Memos. *VLDB J.* **2009**, *18*, 719–738. [CrossRef]
18. Lee, M.L.; Hsu, W.; Jensen, C.S.; Cui, B.; Teo, K.L. Supporting Frequent Updates in R-trees: A Bottom-up Approach. In *Proceedings 2003 VLDB Conference*; Freytag, J.C., Lockemann, P., Abiteboul, S., Carey, M., Selinger, P., Heuer, A., Eds.; Morgan Kaufmann: San Francisco, CA, USA, 2003; pp. 608–619. [CrossRef]