A Survey of Spatio-Temporal Big Data Indexing Methods in Distributed Environment

Ruijie Tian, Huawei Zhai, Weishi Zhang, Fei Wang, and Yao Guan

***Abstract*—With the widespread use of mobile and sensing devices, and the popularity of online map-based services, such as navigation services, the volume of spatio-temporal data is growing rapidly. Conventional big data technologies in existing distributed systems cannot effectively process spatio-temporal big data with temporal continuity and spatial proximity. How to construct an effective index for the application requirements of spatio-temporal data in a distributed environment has become one of the hotspots of spatio-temporal big data research. Many spatio-temporal indexing methods have been proposed to support efficient query processing of spatio-temporal data. In this article, the various spatio-temporal big data indexing methods proposed by domestic and foreign researchers from 2010 to 2020 are classified and summarized according to the distributed environment and application background, and the hot issues that need to be paid attention to in the future are proposed according to the changes in application requirements**

***Index Terms*—Big Data, distributed system, spatio-temporal index.**

I. Introduction

With the maturity and widespread use of perception technology, a large number of records containing temporal and spatial marker information are generated, which are called spatio-temporal data [1]. Spatio-temporal data present multisource, heterogeneous, and multidimensional scale characteristics. Space and time are the basic attributes of spatio-temporal data and the basic characteristics of spatio-temporal data processing. With the rapid development of mobile Internet, location services and other technologies and the popularization of mobile devices, e.g., traffic trajectories [2], social media [3], remote sensing image [4], [5], climate observation [6], and other data containing spatio-temporal information rapidly accumulate, are forming a special kind of spatio-temporal big data [7]. In a general sense, spatio-temporal big data refers to the massive spatio-temporal data collection that is difficult to carry out data management, scientific computing, and value discovery within an acceptable time frame by using existing theories, methods, technologies, and tools.

Spatio-temporal big data contains fund of knowledge value. However, due to the large volume of spatio-temporal data, high computational complexity, and the large amount of time spent in complex spatio-temporal queries, it becomes important to support high-performance queries on spatio-temporal data. Distributed spatio-temporal databases provide a variety of spatio-temporal indexes and queries to achieve fast data retrieval, but the "weak" scalability of index schemas and the "strong" limitations of strict database specifications make them unsuitable for processing spatio-temporal big data. Distributed computing technology achieves high scalability and excellent performance through its computing power, and it has spawned the development of modern parallel processing systems, e.g., Hadoop, HBase, and Cassandra, based on MapReduce. They offer excellent scalability and high-performance computing capabilities compared with traditional centralized systems that process large amounts of data. Recently, distributed computing systems and NoSQL databases have been widely used for indexing of structured, semistructured, and unstructured spatio-temporal data.

Although the distributed system solves the storage management problem of spatio-temporal big data to a certain extent, however, the input data are not organized by spatio-temporal associations, which means that spatio-temporal properties are not considered when deciding where each record is physically stored. This results in poor retrieval performance for applications that are sensitive to spatio-temporal correlations. Therefore, establishing a spatio-temporal index is an effective way to improve the access efficiency of spatio-temporal big data. In recent years, some surveys have been published on the research progress of spatio-temporal data indexing [8], [9], But it is mainly aimed at the analysis of spatio-temporal indexes under centralized systems, e.g., based on main memory [10]–[15], based on flash extended cache [16], based on disk [17]–[20], based on multicore multithreading [21], [22], and GPU-based spatio-temporal indexing [23], [24]. The spatio-temporal index structure mainly includes B-tree, grid [25], quadtree [26], R-tree [27], and some variants of R-trees [20], [25]. The index structure is simple and easy to maintain, which can meet the high-efficiency retrieval requirements of conventional spatio-temporal applications. However, due to the limitation of single-machine resources, it cannot support the application requirements of low-latency access and high concurrent access to spatio-temporal big data. In a distributed system, because the data storage architecture, data

Manuscript received March 15, 2022; revised April 19, 2022; accepted May 13, 2022. Date of publication May 20, 2022; date of current version June 2, 2022. This work was supported in part by the National Key R&D Program of China under Grant 2020YFF0410947, in part by the National Natural Science Foundation of China under Grant 62103072, in part by the China Postdoctoral Science Foundation under Grant 2021M690502, and in part by the Shipping Joint Fund of Department of Science and Technology of Liaoning under Grant 2020-HYLH-50. *(Corresponding authors: Huawei Zhai; Weishi Zhang.)*

The authors are with the Information Science and Technology College, Dalian Maritime University, Dalian 12421, China (e-mail: trj@dlmu.edu.cn; zhw@dlmu.edu.cn; teesiv@dlmu.edu.cn; feiwang@dlmu.edu.cn; 1961842416@qq.com).

Digital Object Identifier 10.1109/JSTARS.2022.3175657

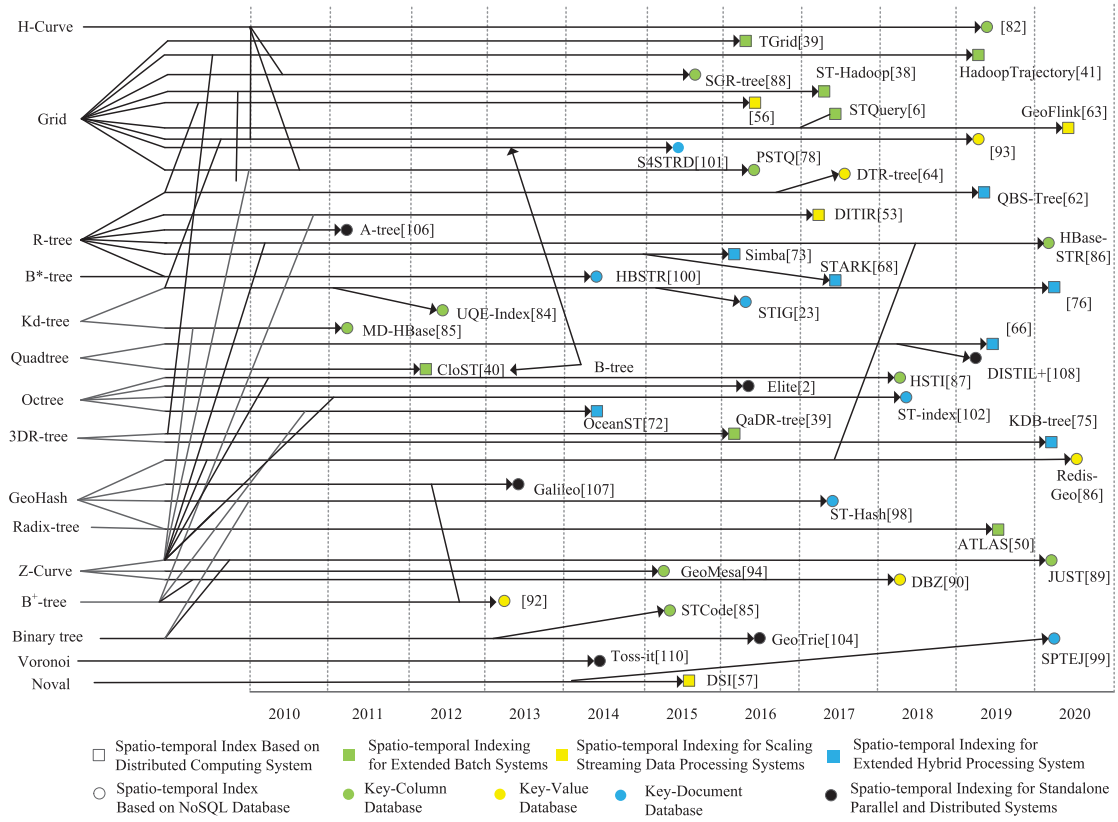

Fig. 1. Spatio-temporal indexing method (2010–2020). The cross lines represent the relationship between a new spatio-temporal index structure and the original index structure it has evolved from.

management model, and data processing method are very different from the centralized system, the indexing technology cannot be easily transplanted into the distributed system. Therefore, the distributed elasticity for spatio-temporal big data. The indexing method has become the main research hotspot at present.

The research scope of this article consists the papers about spatio-temporal big data indexing published between 2010 and 2020. The main literatures come from VLDB, TKDE, ICDE, SIGMOD, CIKM, MDM, and other journals and conferences (including but not limited to SCI and EI). According to the research content, the evolution process of different index modes is classified and summarized. As shown in Fig. 1, the lines represent the relationship between an original index structure evolving into a spatio-temporal index structure.

The batch processing system is suitable for application scenarios, where the storage is first and then the calculation is performed, the real-time requirements are not high, and the accuracy and comprehensiveness of the data are more important. The spatio-temporal index structure often chooses R-tree, 3DR-tree, and Quadtree to form a global–local index structure. Streaming data processing systems no longer store data, but directly perform computations in memory after streaming data are accessed, which is suitable for high real-time and low-precision application scenarios. The spatio-temporal index structure often chooses B^+-trees and grids. The hybrid processing system can meet the data calculation requirements of various application scenarios at different stages, and the spatio-temporal index structure often chooses Quadtree, Kd-tree, and R-tree. The spatio-temporal index based on NoSQL database focuses on the *RowKey* design, and the method adopted is mainly a one-dimensional (1-D) linear expression, e.g., GeoHash [28], Z-Curve [29], and H-Curve [30]. Independent distributed systems are based on peer-to-peer (P2P) network architecture, and octrees and R-trees are often selected as local indexes. In recent years, the number of studies related to spatio-temporal big data indexing has been increasing year by year, and the research on spatio-temporal big data indexing has become one of the research hotspots in the field of spatio-temporal big data.

This article takes the spatio-temporal big data indexing method as the core, systematically sorts out and analyzes the current research and development status of spatio-temporal big data indexing methods, and according to the existing spatio-temporal big data index construction platform, the spatio-temporal big data-oriented spatio-temporal index can be divided into three categories:

1) spatio-temporal index based on distributed computing system;
2) spatio-temporal index based on NoSQL database;
3) spatio-temporal index based on standalone distributed system.

The rest of this article is organized as follows. Section II provides an overview of existing spatio-temporal indexes based on distributed computing systems. Section III outlines spatio-temporal indexes based on NoSQL databases. Section IV

TABLE I
OVERVIEW OF SPATIO-TEMPORAL INDEXING BASED ON DISTRIBUTED COMPUTING SYSTEM

Platform	Prototype	Partitioning	Indexing	Data skew	Queries
Hadoop	ST-Hadoop [38]	Multi-level temporal partitioning	Temporal hierarchy index of spatial indexes at multiple levels of temporal resolution	✓	Range queries, *k*-NN query, joins
	HadoopTrajectory [41]	MBR-based grouping and partitioning	Global index in the form of 3D Grid or 3DR-tree	–	Range queries
	QaDTree [39]	Quadtree-like partitioning	Global/local index: Quadtree-like index/3DR-tree	✓	Spatio-temporal range queries
	TGrid [39]	Grid-based partitioning of spatial dimensions	Global/local index: One-dimensional time index/ Grid	–	Spatio-temporal range queries
	STQuery [6]	Grid-based partitioning	The index structure is designed as a relational database table structure	–	Range queries
	CloST [40]	Hierarchical Partitioning	Hierarchical index: level-1: hash(oid)+coarse ranges(Time)/level-2: space Quadtree/level-3: One-dimensional time index	–	Range queries
	ATLAS [50]	GeoHash Partitioning	Spatial index: a radix tree over geohashes temporal index: a B^+-tree with block timestamps	–	Spatial and temporal queries
Storm	DITIR [53]	Z-order Partitioning on x and y	Z-order and B^+-tree	✓	Real-Time spatial and temporal queries
	Work [56]	Single Dataset: Field grouping and partitioning Joins for Moving Objects: Grid partitioning	Single Dataset: Grid, R-tree Joins for Moving Objects: Grid, R-tree, Hash Table	–	Range queries, *k*-NN query, spatial joins
	DSI [57]	One-dimensional data partitioning	Dynamic Strip Index	✓	*k*-NN query
Spark	DTR-tree [64]	One-dimensional data partitioning	Global/local index: R-tree/ R-tree	✓	Similarity search
	STJoins [66]	Data (re-)partitioning based on Quadtree decomposition	Equi-sized splitting of whole dataset and local Quadtrees	–	Spatio-temporal join
	DITA [67]	Group trajectories based on first and last point and use STR to partitioning	Global/local index: Build an R-tree on the MBR of the first and last points respectively/a variant of trie-based index on selected point	–	Similarity search, similarity join
	STARK [68]	Spatial-only	R-trees	–	Spatio-temporal range queries and joins
	OceanST [72]	Hierarchical Partitioning	Global/local index: level-1: hash(oid)+coarse ranges(Time), level-2: space Quadtree, level-3: One-dimensional time index/3D Quadtree, B^+-tree	–	Spatio-temporal aggregate queries
	Simba [73]	STRPartitioner (sampling and STR)	Global/local index: indexRDD	–	Range query, *k*-NN, distance join, *k*-NN join
	Work [74]	Data (re-)partitioning based on Kd-tree	Global/local/Rdd index: Kd-tree/Hash Table/Kd-tree	–	Temporal and spatial query
	KDB-tree [75]	KDB-tree partitioning (sampling and KDB-tree) and data repartitioning based on the key of pair	R-tree and KDB-tree	✓	Spatio-temporal range queries
Flink	QBS-Tree [62]	Improved STR partitioning	QBS-tree	–	*k*-NN query
	GeoFlink [63]	Grid-based partitioning	Grid	–	Range, *k*-NN and join queries

outlines spatio-temporal indexes based on independent distributed systems. Finally, Section V concludes this article and outlooks a preliminary discussion on future research directions worthy of attention.

II. SPATIO-TEMPORAL INDEX BASED ON DISTRIBUTED COMPUTING SYSTEM

This type of spatio-temporal index is built on the existing distributed computing system, and the main advantage of this method is to inherit the scalability and fault tolerance of the underlying big data processing system. Distributed computing systems mainly include batch processing systems, stream data processing systems, and hybrid processing systems [31]. Batch processing systems (e.g., Hadoop [32]) are mainly used to process static spatio-temporal data and return the results after the calculation is completed. The calculation process takes several minutes or even hours, so it is not suitable for processing tasks that require high real-time performance. Streaming data processing systems (e.g., Storm [33] and S4 [34]) perform calculations on spatio-temporal stream data and process the data with minimal delay. Hybrid processing systems (e.g., Spark [35] and Flink [36]) can handle both batch and stream processing workloads. This section will focus on the related research results of extending these three types of systems to build spatio-temporal indexes. Table I provides an overview of spatio-temporal indexes based on distributed computing systems.

A. Spatio-Temporal Indexing for Extended Batch System

The big data batch processing system is suitable for application scenarios with low real-time requirements and high requirements for data accuracy and comprehensiveness. The typical representative system is Hadoop. As the basic platform of big data, Hadoop is widely used in the storage and processing of spatio-temporal big data. Extending Hadoop's spatio-temporal

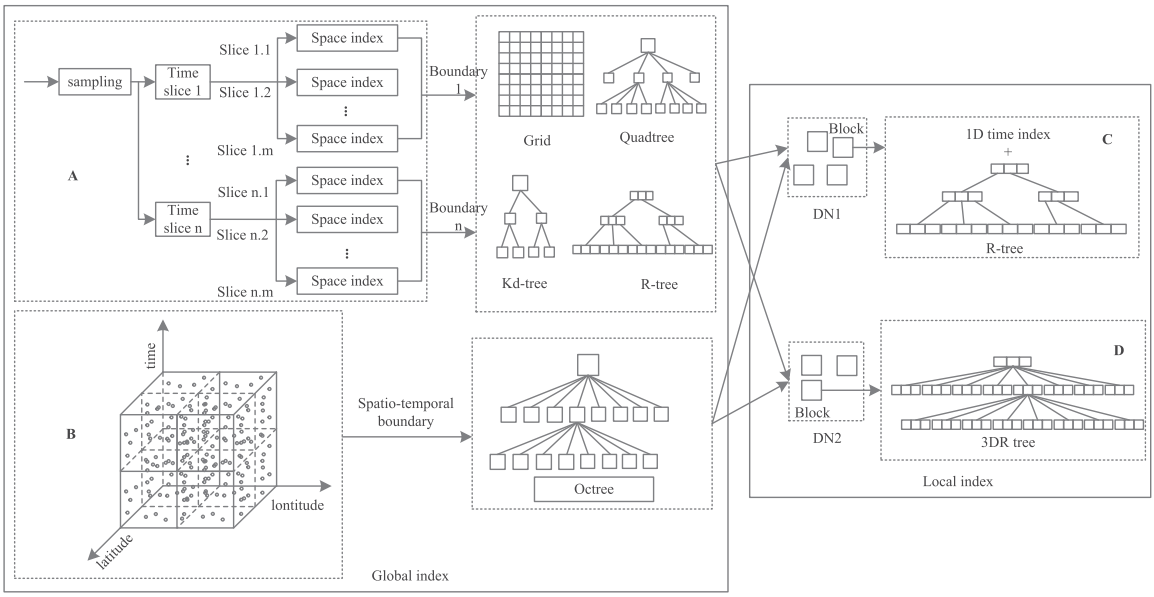

Fig. 2. Spatio-temporal index structure under the Hadoop framework.

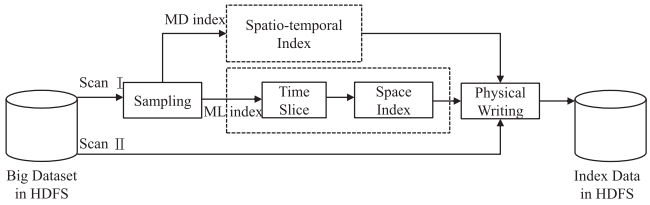

Fig. 3. General index flowchart over batch system.

index is mainly used to index static spatio-temporal objects (e.g., STQuery [6], ST-Hadoop [38], QaDR-tree [39], and CloST [40]) and moving object history track index (e.g., HadoopTrajectory (HT) [41]). It can satisfy high throughput while requiring low latency of query response. Use batch index creation method. Data import and index creation use two MapReduce tasks, respectively. The MapReduce program is simple and easy to implement, but the program running time is expensive. The spatio-temporal index structure under the Hadoop framework is shown in Fig. 2. There are two types of global indexes: multilevel index (ML index) A and multidimensional index (ML index) B. ML indexes usually take the time dimension as the first level, select the appropriate time granularity to process the dataset slices, and the spatial dimension as the second level. Spatial indexing technology mainly includes grid, quadtree, Kd-tree, R-tree, and some variant structures of R-tree. Local indexes are mainly based on 3DR-tree [46] and BR-tree. In this section, we will focus on the main spatio-temporal indexes that scale Hadoop. The general index flowchart over batch system is shown in Fig. 3.

ST-Hadoop [38] is a spatio-temporal big data management framework that extends SpatialHadoop [47], [48]. It is used to store, index, and query spatio-temporal big data. ST-Hadoop extends the Hadoop system by injecting spatio-temporal awareness in the spatio-temporal query language layer and spatio-temporal indexing layer. ST-Hadoop is designed as a static hierarchical index structure with time slicing and then spatial indexing. The creation of the spatio-temporal index executes the following stages:

1) sampling;
2) time slice;
3) spatial index;
4) physical write.

During the sampling phase, MapReduce scans the spatio-temporal data, takes random samples, and stores them in main memory. In the time slicing phase, the random sample is divided into multiple time slices by applying the time partition slicing or data partition slicing algorithm. Time partition slicing is to divide a random sample into multiple nonoverlapping slices according to a given time granularity (year, month, week, and day), the time range of the slices is fixed, and the number of data points contained in the slices is not exactly the same. A data partition slice is where each slice contains roughly the same number of data points, while the time ranges of the slices may not be the same. In the spatial indexing stage, a spatial index is built for each time slice using traditional spatial indexes already existing in SpatialHadoop (e.g., grid, R-tree, Quadtree, and Kd-tree). The spatio-temporal boundary is stored on the ST-Hadoop master node as metadata, and the metadata format is $\langle \text{id}, \text{MBR}, \text{interval}, \text{level} \rangle$. During the physical write phase, data records are allocated to overlapping partitions according to the spatio-temporal boundaries stored on the master. When doing a spatio-temporal query, the operator first filters by time, obtains the corresponding time slice, and then performs spatial filtering.

HT [41] is a spatio-temporal big data processing system with built-in spatio-temporal data types and spatio-temporal operations on Hadoop, and supports storage, indexing, and querying of moving objects and their trajectory data. HT supports two

index structures: space-based indexing and data-driven indexing. In space-driven indexing, the selection grid divides the 3-D extent of the input data into cubes, each of which maps to one or more files. The grid structure divides a trajectory into multiple consecutive grid cells. In the data-driven index, select 3DR-tree to partition the large data file of the moving object into multiple blocks, and the block size is set to 4 M, which not only preserves the movement Object Semantics and Structure (for example, a moving object must not be split into multiple partitions) and unnessary to traverse multiple objects in a block file. The spatio-temporal bounding box of the trajectory is stored in the 3DR-tree, and the index information is stored in the master node. When creating an index, you need to specify whether the index type is a track or a moving object. In the track type, the 3-D minimum bounding rectangle (3-D MBR) of each track is stored in the index, whereas in the moving object type, the 3-D MBRs of all trajectories belonging to the same moving object are merged into one 3-D MBR and stored in the index. HT supports single query window, multiple query windows, specific moving objects, or specific trajectories for query.

QaDTree&TGrid [39] are distributed indexing method for querying historical spatio-temporal data. It is designed to perform spatio-temporal range queries on different distributed spatio-temporal data under HDFS. QaDR-Tree is an ML index structure for nonuniformly distributed spatio-temporal data, which consists of a global index and a local index. The global index is built based on octrees [49]. It splits the spatio-temporal data into blocks, and the block size is set to 60 MB, which is smaller than the default size of 64 MB for Hadoop data blocks, and the purpose is to allocate additional storage space for local indexes to be stored in Hadoop data blocks along with the spatio-temporal data. A local index is a 3DR-tree, and a 3DR-tree is an R-tree with a time dimension added. The leaf node points to the local index tree. When performing a spatio-temporal range query, the 3-D quadtree quickly finds the node position of the data block intersecting the spatio-temporal window; then, the 3DR-tree in each data block is used to filter the final query result. TGrid is an MD index structure for uniformly distributed spatio-temporal data. It adds time dimension to the traditional grid index (the grid is represented as the MBR) to construct the TGrid (the grid is represented as the minimum bounding cube MBC). TGrid first sets a fixed time granularity in the time dimension and then divides the space dimension uniformly. In the local index stage, a 1-D time index is created for the spatio-temporal data of different MBCs, and the local index results are aggregated on the master node to build a global index.

STQuery [6] is a spatio-temporal indexing method that uses mapreduce to efficiently retrieve and process array climate data, and supports storage in HDFS in the form of raw files. STQuery uses a grid to map the logical multidimensional array data model of a file into a MapReduce key-value pair structure. The spatio-temporal index information is stored in a relational database. The index structure is designed as a relational database table structure, contains gridId, startByte, endByte, nodeList, and fileId. StartByt and endByte record the starting and ending positions of the grid where gridId is stored on HDFS, nodeList represents the node location of grid storage, and fileId records the file to which the grid belongs and the file compression type. Based on the index, a grid allocation and grid combination strategy is proposed to reduce data transmission across nodes and balance the workload of nodes to optimize MapReduce computing performance. However, 1) when the array file is uploaded to HDFS in the form of a byte stream, the logical grid data will be divided into two blocks due to the default block size limit, and the data will be transmitted across nodes during the query process, which will affect the performance of MapReduce; 2) the amount of data contained in each grid under the high resolution of space and time will be very large, and a large amount of space and time data will still be scanned when performing range query, and the query efficiency will not be very high.

CloST [40] is a spatio-temporal big data system based on Hadoop. CloST supports single-object $Q(I, S, T)$ queries and full-object $Q(S, T)$ queries, where I represents the object id, S represents the spatial range and T represents the time range. CloST is designed as a hierarchical partition structure, and the original table data are divided into three layers: the first layer roughly divides the data into buckets according to the data oid hash value and rough time range, the second layer divides the buckets into regions according to the fine-grained spatial index, and the third layer uses 1-D time range division to divide the regions into HDFS slice. For hierarchical partitioning, CloST is designed as an ML index structure. The first-level index uses a hash table structure to filter oid and time coarsely, the second-level index uses a Quadtree to divide spatial data, and the third-level index uses a 1-D index structure B^+-tree, etc. CloST designs a file format for HDFS blocks that supports storage management on Hadoop (e.g., RCFile, Parquet). The file format supports intrablock indexing to further optimize search performance and space utilization.

ATLAS [50] is a distributed file system for storing, indexing, and querying spatio-temporal data that extend Hadoop HDFS. ATLAS utilizes the idea of data file partitioning to improve data access efficiency. When a block of data is written to the file system, the GeoHash value of the spatio-temporal objects within the block is calculated, and the spatio-temporal objects are rearranged to match the spatio-temporal boundaries. The spatial index based on radix tree [51] is constructed according to the GeoHash value of the data block. Each node in the tree maintains a list containing a block ID, offset, and length tuple representing the data belonging to the corresponding node GeoHash. Queries on spatial indexes return a superset of the requested data to ensure full coverage. For instance, a query on data with a GeoHash value of "8bce" will return data with an index of "8b." ATLAS maintains a temporal index using a B^+-tree, which has each the start and end timestamps of the block.

To sum up, as shown in Fig. 5, in an MD index, space and time are treated as dimensions in multidimensional space, and there is no difference in the order of processing space and time. In ML index, the processing of space dimension and time dimension is different, and the idea behind this type of index is to build a separate spatial index for each time instance. In addition, for

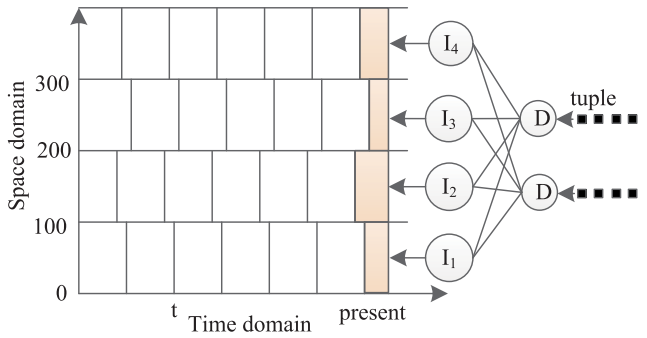

Fig. 4. Global data partition.

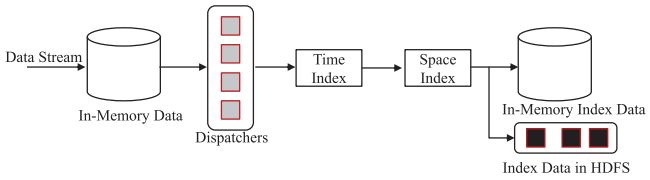

Fig. 5. General index flowchart over stream processing system.

different types of spatio-temporal queries, the data are actually stored in multiple copies, which speeds up the query efficiency, but results in redundancy of multiple copies storage.

B. Spatio-Temporal Indexing for Scaling Streaming Data Processing System

With the continuous generation of spatio-temporal data by intelligent positioning devices, the demand for ingestion and retrieval of streaming data is also growing rapidly. Traditional big data streaming computing no longer stores data but when streaming data arrives. After that, real-time processing is performed directly in memory. The spatio-temporal flow data present the characteristics of real-time, volatile, sudden, disordered, unbounded, and spatio-temporal correlation, which puts forward many new and higher requirements for the system [52]. In 2010, Yahoo launched the S4 streaming system with a symmetric architecture. In 2011, twitter launched the Storm streaming system with a master–slave architecture. These promote the development of big data streaming data processing technology to a certain extent, and also provide the possibility for processing spatio-temporal data streams. The general index flowchart over stream processing system is shown in Fig. 5.

DITIR [53] is a distributed index structure for high-throughput trajectory insertion and real-time trajectory data query. It is built on the distributed data flow system Apache Storm. DITIR assumes that all data tuples are incoming in timestamp order. DITIR uses transceivers to transmit data. The spatial information x and y of the incoming data tuple $\langle x, y, t, e\rangle$ are transformed into $\langle z, t, e\rangle$ by Morton encoding [54], and each data tuple is divided into the insertion server according to the z value of the spatial range. Each insertion server stores its input data tuple in the memory B^+-tree. The key value of the B^+-tree node is the z value of the data tuple. When the capacity of the B^+-tree reaches a predetermined threshold, the insert server persists the B^+-tree as a data block to distributed file system. In order to avoid the huge overhead of node splitting caused by inserting a large amount of data into the B^+-tree, DITIR uses the template-based B^+-tree insertion mode [55]. In template-based index construction, it is assumed that the spatial distribution of the data (i.e., the distribution of z-values) does not change significantly over time. DITIR uses the B^+-tree structure (all leaf nodes are empty) of the previous data block as a template for the next data block to use. The query server maintains metadata about blocks in a distributed file system to improve DITIR search performance. The metadata server uses R-trees to store spatial extent information for HDFS data blocks, and the query coordinator converts user queries into independent subqueries that are executed in parallel on the insertion server and query server.

Zhang *et al.* [56] aimed at the real-time spatial query under the high dynamic data update of single moving object and multiple moving objects, and reduced the time complexity of data update by modifying the traditional spatial index. Single moving object query first groups spatial data into Storm bolts by the FieldsGrouping method according to the moving object ID; second, builds distributed indexes (e.g., R-tree and grid) in each bolt instance, and obtains local results through the main memory spatial query algorithm; and finally, aggregate the local results to get the global results. Single moving object query distributes update messages of spatial information to executors based on point ID, thereby reducing the pressure of frequent updates and real-time calculation and analysis of a large number of moving objects. Multimoving object index is a distributed hierarchical index structure. The spatial range is divided into uniform grids, which is used as the main index. Two sets of secondary indexes (e.g., R-trees, hash tables, and quadtrees) are maintained for each grid to organize spatial data from the two datasets, respectively, and a result table to store spatial join results.

There is also a special real-time moving object query, that is, the *k*-NN query of real-time moving objects. This query only needs to maintain a list of moving objects, and there is no throughput constraint caused by index structure and data insertion, but it will involve the problem of throughput between nodes: mass communication and transfer of information. Since the moving objects are constantly updated, the spatio-temporal flow is an unbounded flow type, and the number of moving objects is theoretically infinite, but in the actual environment, the value is approximately unchanged, only a limited list of moving objects and index queries need to be maintained.

When implementing *k*-NN query of real-time moving objects in a master–slave architecture, the master node is responsible for storing, maintaining index information, and distributing query requirements, but it will face the following problems.

1) For master–slave architecture, nodes may be distributed to different servers, resulting in frequent interaction between master and slave servers, increasing the communication burden.

2) The master node is responsible for distributing queries to the slave nodes; however, the frequent movement of objects can easily make the master node into a performance bottleneck.

DSI [57] is a distributed index structure based on the Apache S4 system, which supports *k*-NN queries for real-time moving objects. DSI divides the 2-D space into two sets of nonoverlapping strips (vertical strips and horizontal strips). The index structure is designed as $\{id_i, lb_i, ub_i, \Gamma_i\}$, id_i is the unique identification of the strip, lb_i and ub_i are the upper and lower boundaries of the strip, and Γ_i is the list of moving objects. The number of objects in a strip has a lower threshold ξ and an upper threshold θ. When the number of objects in a strip exceeds the upper threshold θ, the strip is split into two strips. Conversely, when the stripe has fewer objects than the lower threshold ξ, stripes will attempt to merge with adjacent strips. DSI indexing tasks on S4 involve two types of PEs, EntrancePE and IndexPE. EntrancePE stores the list of moving objects L_V and L_H. IndexPEs are assigned to different nodes in the cluster through a hash function, and each IndexPE manages a stripe. Using DSI for *k*-NN query, first identify candidate strips according to the spatial constraints of query q to calculate the local *k*-NN result set; then, aggregate all local *k*-NN result sets into a global *k*-NN result set.

Most existing grid-based *k*-NN search methods iteratively expand the search area to identify *k*-NN, when adopted in a master–slave architecture, let the master node maintain partition information (e.g., cell boundaries) and the slave nodes maintain the index of each cell. In this case, each iteration requires a round of communication between the master and slave nodes holding the relevant cells, which is an expensive operation compared to other costs. When adopted in a symmetric architecture, the abovementioned problems can be avoided, but the required number of iterations cannot be guaranteed, resulting in unpredictable query performance. Tree-based indexes (e.g., R-trees, Kd-trees, B^+-trees, and TPR-trees) involve frequent splitting and merging of nodes, have high maintenance costs, and are not suitable for processing queries of real-time moving objects, and many previous studies [58], [59] had paid attention to this problem.

"Strong" real-time requirements for *k*-NN queries for real-time moving objects are as follows:

1) indexes can be easily partitioned and distributed to different servers in the cluster;
2) the index can be efficiently updated with the continuous movement of the object;
3) the index supports an efficient *k*-NN search algorithm with fewer iterations.

DSI can meet this requirement better than existing grid-based and tree-based indexes, but 1) a set of data needs to maintain two sets of indexes. When updating a moving object, two sets of indexes will be updated at the same time, and the index maintenance cost is high. 2) When executing *k*-NN query, the same *dk*-NN operation needs to be performed for each two indexes respectively, although iterative. The number of queries can be set in advance, but the computational complexity is high.

For the ingestion of spatio-temporal stream data, real-world applications usually require extremely high tuple insertion throughput and real-time response to range queries on a specified domain. Therefore, real-time indexing of spatio-temporal stream data is performed in the traditional streaming computing mode, and range query processing faces the following two main challenges.

1) How to efficiently maintain real-time data range queries while also supporting efficient insertion of spatio-temporal stream data?
2) How to efficiently index data tuples on both the temporal and spatial domains while keeping new incoming tuples immediately visible to queries as they arrive?

The usual solution is to use a global data partitioning model, assuming that the streaming data arrives approximately in order. Specifically, the spatio-temporal space is divided into cubes, which are called data regions, and the incoming data tuples are stored in their corresponding data regions when they arrive. For intuitive display, Fig. 4 shows a 2-D schematic diagram of the spatio-temporal space. Given any query with a specific query region, this partitioning mode can effectively speed up query execution by skipping data regions that do not have any overlap with the query region. More importantly, since data tuples arrive at the system roughly in the order of their timestamps, newly arrived tuples are always inserted into the data region with the latest timestamp, i.e., the rightmost data region (shaded) in Fig. 4, while not a historical data area. Therefore, the physical partitioning between recent and historical data in the global data partition mode avoids costly global data merging, enabling high-throughput data insertion, compared to other data structures (for example LSM tree [60]) that mix historical and new data.

An index server is maintained for each unique key in the spatial domain, so that newly arriving tuples of different keys can be injected into the system in parallel. To implement time partitioning, each index server accumulates received data tuples in memory and flushes them to the file system as immutable data blocks when the size of in-memory tuples reaches a predetermined threshold (e.g., 16 MB). Refresh operations on different index servers are asynchronous, so the time boundaries between different key intervals are not aligned, as shown in Fig. 4. In order to further improve the query efficiency, it is necessary to index the data tuples according to the key and time domain in each data area. Based on the following two considerations, it is often chosen to build a B^+-tree on the key domain.

1) Spatio-temporal application queries usually involve low selectivity in the key domain and high selectivity in the time domain.
2) The cost of inserting data into other 1-D index structures is significantly higher than that of B^+-trees.

If the query is selective in both fields, a technique, e.g., Z-order, can be applied to convert the two fields to 1-D integers before building a B^+-tree index. For the problem that the newly incoming tuple is immediately visible, the real-time incremental index construction method can be adopted. When new data arrives, the B^+ tree is used to find the storage location of the data. The index is modified while writing the data, so that the index

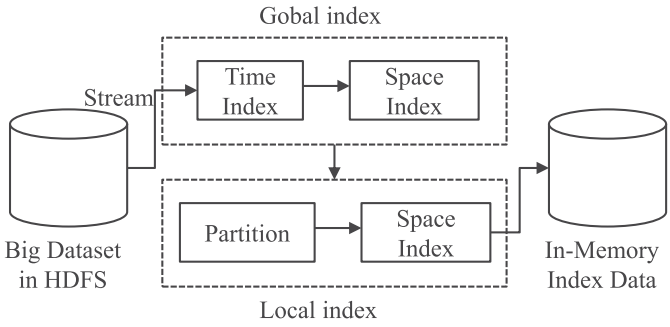

Fig. 6. General index flowchart over hybrid processing system.

can reflect in time. Current latest data status. By delaying disk write operations by caching, the efficiency of data processing can further be improved.

To sum up, from the perspective of maintaining the spatial location information of moving objects, the maintenance cost of tree-based indexes is high in the case of frequent updates, especially when deployed in streaming data processing systems. Grid indexing usually involves iteratively expanding the search area to identify the set of cells contained in the grid. In general, the number of such iterations is indeterminate. More importantly, in streaming data processing systems, grid indexing can lead to excessive communication between nodes in the cluster, thereby reducing the performance of query processing. Considering the communication cost, maintenance cost, and index performance, whether it is a tree-based index or a grid index, the two existing main index types for real-time spatio-temporal stream range query and *k*-NN query cannot be directly used for stream processing system.

C. Spatio-Temporal Indexing for Extended Hybrid Processing System

The hybrid processing system is another mode of real-time spatio-temporal stream processing, and the typical representative systems are Apache Spark and Apache Flink. Apache Spark uses distributed memory for data computing to quickly respond to queries and return analysis results in real time. Spark provides a higher level API than Hadoop, and the same algorithm runs 10 to 100 times faster in Spark than Hadoop [61]. Apache Flink is a scalable platform for batch and streaming data processing. Flink itself does not support the efficient processing of spatial data streams. In view of this, Flink-based spatio-temporal index research is roughly divided into two categories: building spatio-temporal indexes on Flink (e.g., QBS-Tree [62]) and extending Flink's spatio-temporal indexes (e.g., GeoFlink [63]), both of which support continuous queries on real-time spatio-temporal streams. The general index flowchart over hybrid processing system is shown in Fig. 6.

1) Apache Spark-Based Spatio-Temporal Index: In specific application scenarios with high real-time requirements (e.g., real-time location recommendation and moving object behavior analysis), the system is required to provide second-level real-time data stream processing capabilities. Obviously, the disk-based Hadoop cannot satisfy real-time requirements. The Spark-based streaming framework Spark Streaming can solve such problems. The principle is as follows: the continuous stream data are represented as a highly abstract discrete stream packaged into RDDs, and then the RDD is operated in a batch-like manner. Spark Streaming is still a batch method in essence, but due to the characteristics of Spark's in-memory computing, it has a faster processing speed and, thus, achieves quasi-real-time performance. Spark runs on the Hadoop cluster and accesses the distributed file system HDFS, and can also process structured data in Hive and streaming data in Flume and Kafka. Spark RDD provides the possibility of explicitly cache data between iterations to improve computational fault tolerance.

Distributed trajectory R-tree (DTR-tree) [64] is an R-tree index implemented on Apache Spark. It is widely used for trajectory and active trajectory query, e.g., DMTR-tree [65], which supports skyline query of active trajectories with the help of DTR-tree and inverted list. DTR-tree index's trajectories and active trajectories are using a distributed R-tree based on the spatial properties of trajectories. DTR-tree is a global index and a local index structure. The global index adopts an R-tree to provide partitioning of the index track data. The leaf nodes of the R-tree represent the sub-R-trees stored in the distributed nodes. The dataset D is divided into N partitions using a balanced partition strategy, and an R-tree local index is built on each worker node. During range query, the 2-D R-tree index track is used in DTR-tree according to the spatial position information of the track. First, the partition id that overlaps with the query point q is found, and the spatial information is trimmed in the local index corresponding to the partition id to obtain the final result set. The DTR-tree index is established in memory, and when the task is executed, it will be released with the memory, and the cost of indexing each time is very high.

ST-Joins [66] is an extension of the spatio-temporal connection under Apache Spark, and the spatio-temporal connection methods, Broadcast join and Bin join, are designed to adapt to the connection query of datasets of different sizes. The index construction order is space first and then time. Broadcast join is suitable for at least one dataset that can be completely stored in Spark executor memory, the remaining data are distributed to each executor, a local quadtree is established in each executor memory, and all local quadtrees are assembled to form a global quadtree. Bin join performs regular grid or adaptive grid (based on quadtree) division according to the spatial distribution of the dataset to generate grid cell bins, group the data in the same bin together, and construct a quadtree index for the grouped data to further filter the spatial information. Since the data skew leads to the partition bin load, the time dimension is introduced to distribute the spatially dense regions uniformly in time. Adaptive grid division is performed on the dataset, although the grid data distribution is guaranteed to be balanced, but the time complexity of the calculation is increased.

DITA [67] is a research prototype for extending Spark for memory trajectory analysis. DITA uses global/local indexes and proposes a pivot point-based approximate representation technique for trajectories. The STR algorithm is used to divide the data, and the selected trajectory points are operated, that is,

the starting and ending points of each trajectory. Trajectories are grouped based on their first point, and then subgroups are created based on the grouping of the last point. Then, the global indexing system consists of two R-trees, one based on the first point of the MBR and the other based on the last point of the MBR. The local index is a variant of the trie-based index, which is built on the pivot of the trajectory above the point. At the algorithm/processing level, DITA adopts a filter-optimization paradigm in order to efficiently handle similarity search and similarity connection.

STARK [68] is a framework for processing connection query, *k*-NN query, and range query based on Spark's support for spatio-temporal data. STARK supports fixed grid partitioning and binary space partitioning. The fixed grid partition divides the data space dimension into multiple intervals to form a grid cell with equal dimensions, then calculates the partition boundary, traverses the entire dataset, and calculates the grid cell to which each piece of data belongs according to the spatio-temporal information. STARK supports real-time indexing. When executing a partition query, an R-tree index is established for the partitioned data, and finally, the index is used to find the query object. The live indexing method takes a preorder traversal of the tree and an optional partition type as parameters. STARK supports three spatio-temporal operators: contains, is contained, and intersects. DBSCAN clustering based on spatial partitioning is also supported. Back up the points within the distance partition boundary ϵ to adjacent partitions, perform local clustering on each partition in parallel, and merge the local clusters using the backup points in the merge operation. When using real-time R-tree-based indexing and binary space partitioning, when configured, STARK outperforms GeoSpark [69], [70] and SpatialSpark [71].

OceanST [72] is a Spark-based spatio-temporal mobile broadband data (MBB) exact and approximate spatio-temporal aggregation query framework. OceanST adopts a three-level hierarchical partitioning strategy. First, the MBB data are divided into multiple first-level partition buckets according to the hash value of the user id and the coarse-grained time range; second, each first-level partition is divided into multiple; finally, each secondary partition is divided into multiple tertiary partition blocks according to the fine-grained time range, and each bucket can perform data recording and storage optimization. OceanST supports intrablock indexing and inverted indexing. The block file data are grouped according to the user id, the data in each group are sorted by time, and the B^{+}-tree block index is established for the user id. Select the octree associated with the attribute of interest in the block file to create an inverted index. OceanST's hierarchical partition design and index structure design enable it to meet precise and approximate spatio-temporal aggregation queries in different application scenarios. Exact query calculates all leaf nodes covered by the spatio-temporal query range. Approximate query randomly selects B leaf nodes from the leaf nodes covered by the spatio-temporal query range to replace by setting a random sampling threshold B. The approximate single spatio-temporal aggregation query executes a single random index sampling algorithm RIS. The approximate multitime-space aggregation query concurrent random index sampling algorithm performs hierarchical sampling and overlapping sample reuse for concurrent spatio-temporal aggregation, but the OceanST index's maintenance cost is high, and the random sampling calculation of aggregate query results will fall into the local optimal problem.

Simba [73] is an extension of Spark designed to provide efficient query and analysis systems for spatial big data. Simba adopts global indexing and local indexing strategies. Global indexes have simple index structures that support 1-D data indexing, e.g., sorted arrays and complex indexing structures, e.g., R-trees or Kd-trees, that support multidimensional data indexing. The global index is stored in the master node memory by default, and also supports persistence to the file system. Simba establishes a custom index (e.g., R-tree) on each partition data as a local index, and introduces IndexedRDD to support fast random access of spatial data in the partition. Simba supports range queries, distance queries, *k*-NN queries, as well as distance join queries and *k*-NN join queries. Simba optimizes spatial queries but does not support spatio-temporal queries.

Hu *et al.* [74] proposed a Spark-based geospatial raster big data processing framework to support indexed queries on raw geospatial data and designed a three-level hierarchical index:

1) a global index, which builds a Kd-tree at the master node to see the physical storage locations of all blocks in the cluster;
2) a local index, which builds a hash table for each variable at each worker node and index all blocks stored in local worker nodes and provide block layout information at byte, block, and file level;
3) RDD index, build Kd-tree to persist and index all memory blocks in each RDD partition to accelerate spatial queries of in-memory data and avoids linear scans.

Hierarchical index information is stored in a distributed file system. The division of global and local indexes reduces the size of the index to be loaded into memory by each node and avoids the transfer of block metadata between master and worker nodes.

KDB-tree [75] proposes a spatio-temporal Ripley's K-function computation framework based on Apache Spark, which aims to rapidly analyze spatio-temporal point patterns. For spatio-temporal data partitioning, a spatio-temporal index based on KDB-tree [76] is designed to accelerate the computation of spatio-temporal Ripley's K function in a distributed environment. KDB tree construction consists of three steps:

1) sampling, randomly selecting spatio-temporal points and sending them to the cluster master node;
2) the master node establishes a spatio-temporal index based on the KDB tree on the sampling point, and the number of partitions is determined by the construction parameters of the tree;
3) the index is sent to the worker node, and the worker queries the leaf node to which each spatio-temporal point belongs and constructs key-value pairs, which will have key-value pairs with the same key value are divided into the same partition.

For the fast acquisition of spatio-temporal point pairs, it uses a 3DR-tree to reduce the number of redundant spatio-temporal points traversed inside the spatio-temporal Ripley's K function.

2) Apache Flink-Based Spatio-Temporal Index: Apache Flink is an open-source system for scalable processing of batch and streaming data. In this processing system, each input tuple acts as both retrieval and update, that is, each input tuple issues a query, gets the result from the data stored in the index, and then inserts the tuple into the index to for subsequent inquiries. After a period of time, the expired tuples are removed from the index to improve query efficiency. In Flink, all tuples with the same key are processed by a single operator instance. The Keyby operator logically partitions stream tuples based on their keys.

Streams can transfer data between two operators in one-to-one (or forwarding) mode or redistribution mode. A pair of first-class patterns preserves the partitioning and order of elements, while redistribution changes the partitioning of the stream. Depending on the selected transformation operator, each operator subtask sends data to a different target subtask. By default, each operator preserves both the partitioning and ordering of their previous operators, thereby preserving source parallelism. However, the key operation will lead to data reorganization and distribution overhead, and data forwarding may lead to load imbalance or even idle CPU cores so that the computing power of the entire cluster cannot be fully utilized. Therefore, in order to ensure the efficient execution of queries, it is necessary to find the right balance between data redistribution and data forwarding. Furthermore, since parallel instances of an operator cannot communicate with each other, it must be up to the user to ensure data locality for each instance.

In the case of frequent updates, tree indexes either need to adjust the structure of the tree frequently or need to redistribute a large amount of data. More specifically, for Flink processing systems, regular tree indexes (e.g., R-tree, Quadtree, and KD-tree) can only meet the requirements of query efficiency, but it does not perform well in update efficiency, especially in the case of a large number of insertions and deletions. Therefore, building a tree-based spatio-temporal index on the Flink processing system needs to satisfy the following characteristics.

1) High query efficiency: It is required to build a well-structured and low-depth balanced tree to achieve maximum query speed.
2) High update efficiency: The tree structure is required to avoid structure adjustment or data redistribution as much as possible.

In addition, grid-based indexes can achieve fast updates. However, query efficiency is much lower than tree-based indexes.

QBS-Tree [62] is an efficient spatial index based on the Flink processing system, inherits and extends B-tree and R-tree, and supports node insertion and deletion operations. QBS-Tree is a self-balancing tree that is equally efficient in the face of spatially unbalanced index objects. QBS-Tree draws on the idea of R-tree to aggregate data points with similar distances, and represents the index objects of these data points as the center point of a 2-D rectangle to reduce the overlapping area between the MBR of different index objects. When a new index object is inserted, if the number of leaf node index items exceeds the threshold M, the node executes the splitting algorithm of the improved STR, the leaf node is divided into four child nodes, and the QBS-Tree is adjusted to be a balanced tree. When the index object is deleted from the leaf node, when the number of index items of the leaf node is less than the threshold M, the subnodes of this layer are merged to form a new leaf node, and the QBS-Tree is readjusted to a balanced tree. QBS-Tree novelly proposes a delayed update mechanism, which can optimize the update operation and avoid unnecessary structural adjustment of the QBS tree. Due to the rapid arrival and termination of streaming data, the cost of maintaining QBS-Tree to complete query and update tasks is high.

GeoFlink [63] is a streaming computing framework that extends Flink to support continuous queries on spatio-temporal data streams. GeoFlink introduces a grid-based spatial index. The grid G is constructed by dividing a 2-D rectangular space into grid cells of length l, where $c_{x,y}$ represents a grid. GeoFlink assigns each stream tuples a unique key according to the grid cell to which $s.x$ and $s.y$ belong. We aim to assign all tuples with the same key to the same partition while optimizing the spatial distribution of objects evenly in a distributed system. Quickly filter query objects by key when performing spatial queries. GeoFlink's special spatial index mode makes it unnecessary to maintain an additional data structure to store index information. Therefore, when the stream tuple expires or a new data object is received, the index structure does not need to be updated, resulting in fast indexing and high memory efficiency.

D. Summary

This article mainly discusses extending spatio-temporal indexes under existing distributed computing systems. These indexing methods not only inherit the fault-tolerant and highly scalable characteristics of big data processing systems but also make full use of the computing power of the system to satisfy spatio-temporal queries in different scenarios. However, as the spatio-temporal application scenarios gradually become real-time, the traditional single index structure is difficult to effectively support the actual demand. Therefore, in the face of spatio-temporal big data, the spatio-temporal indexing technology based on distributed computing systems can summarize three development trends.

1) Specialization of index technology: In order to reduce query cost and improve response efficiency, spatio-temporal indexes need to get rid of the traditional centralized index system and tend to specialized index technology.
2) The index structure is diversified: Mature centralized indexing technology has been gradually integrated into distributed computing systems. However, the uniqueness of the distributed architecture and the rich query operations of spatio-temporal big data have unique requirements for index structure design. ML index, MD index, and combined index structure have gradually become the mainstream technologies of spatio-temporal big data indexing.
3) Real-time index query: In the context of spatio-temporal big data and practical application requirements, as a spatio-temporal index supplement for the extended batch system, it aims to shorten the query response time of PB-level data to seconds and real-time indexes that can

TABLE II
OVERVIEW OF SPATIO-TEMPORAL INDEXING BASED ON NOSQL DATABASE

Platform	Prototype	Partitioning	1D Mapping	Data type	Data skew[d]	Adaptive[e]	Queries
Cassandra	PSTQ [78]	Murmur3Partitioner	Z-order	ST[a]	–	–	Predictive k-NN, range
	MLS3 [82]	Quadtree, Hilbert	Hilbert	ST	–	✓	Spatio-temporal range
HBase	UQE-Index [83]	Z-ordering	Z-order	ST	✓	✓	Range
	MD-HBase [84]	Z-ordering	Z-order	MD[b]	–	–	Range and k-NN
	STCode [85]	Range binary	Generic	ST	–	–	Range and k-NN
	HBase-STR [86]	GeoHash	GeoHash	ST	–	–	Range, distance join
	HSTI [87]	Equal-size cells	Z-order	ST	–	–	Spatio-temporal k-NN
	SGR-tree [88]	Grid partitioning	Hilbert	ST	✓	–	Range and k-NN
	JUST [89]	GeoHash	Generic[c]	ST	✓	–	Range and k-NN
Accumulo	DBZ [90]	GeoHash	Z-order	ST	–	✓	Range
	Work [92]	GeoHash	GeoHash	ST	–	–	Range
	Work [93]	Grid partitioning	Hilbert	ST	✓	✓	Range
	GeoMesa [94]	GeoHash	Z-order	ST	✓	–	Range
Redis	Redis-Geo [86]	GeoHash	GeoHash	ST	–	–	k-NN
MongoDB	STIG [23]	Kd-tree	GeoHash	MD	✓	–	Spatio-temporal range
	ST-Hash [98]	Z-ordering	Z-order	ST	–	–	Range
	SPTEJ [99]	Octree-based on equi-sized	Z-order	MD	–	–	Spatio-temporal joins
	HBSTR [100]	R-tree	Store-only	ST	✓	–	Range
	S4STRD [101]	Grid	Grid position	ST	–	–	Spatio-temporal Range
	ST-Index [102]	Octree-based on equi-sized	Morton	ST	–	✓	Spatio-temporal range

[a]ST mean spatio-temporal data support. [b]MD mean multidimensional data support. [c]Generic represent a novel 1-D mapping approach. [d]In the case of skewed data, identify the most appropriate 1D mapping for the dataset. [e]Aims to find the best mapping for a given query task.

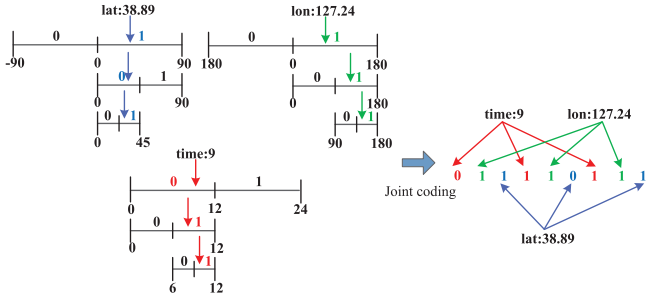

Fig. 7. Spatio-temporal MD index.

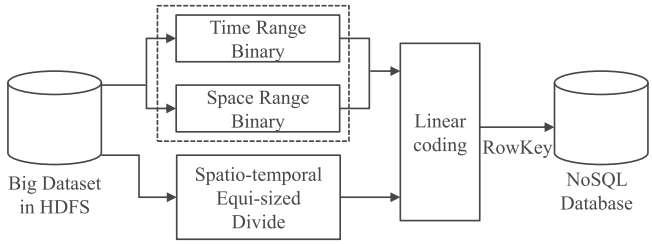

Fig. 8. General index flowchart over NoSQL database.

perform continuous queries are receiving increasing attention.

III. SPATIO-TEMPORAL INDEX BASED ON NOSQL DATABASE

In order to solve the problems of the traditional relational databases in spatio-temporal big data management, high concurrent reading and writing, and expansion. NoSQL database technology has become the technical support for spatio-temporal big data storage management. At present, NoSQL databases usually use the key–value data model to store data. Divided into three data models: key–column, key–value, and key–document [77]. A NoSQL database accesses data based on the key–value model. After the data table is sharded, it is distributed and stored on the server cluster.

There are two structures for existing spatio-temporal indexes based on NoSQL databases. 1) The spatio-temporal multidimensional index, as shown in Fig. 7, is an extension of spatial multidimensional indexes, which regard time as an additional dimension and jointly encode spatio-temporal information into 1-D data. Establish the mapping relationship between *RowKey* and the node where the data block is located, which is suitable for three data models. 2) Joint index, mainly used for key–column data model. The essence of joint index is also an extension of spatial multidimensional index. GeoHash, Z-order, H-Curve, and other technologies are encoded into 1-D data, and the 1-D data are organized in a tree structure to speed up the positioning of spatial data. Second, the mapping relationship between 1-D data and data nodes is established by using the Hash algorithm. The last 1-D data are jointly encoded with time information and other attributes as *RowKey* as the only access entry for data rows in the NoSQL database. The joint index is designed to quickly locate spatial data blocks, and does not do special processing for time information. In order to improve the query efficiency of spatio-temporal big data, some have built a secondary index [78]. Table II provides an overview of different spatio-temporal indexes under NoSQL databases. The general index flowchart over NOSQL database is shown in Fig. 8.

A. Key–Column Database

The key–column data model is mainly from Google's BigTable. The typical representatives are Cassandra [79], HBase [80], and Accumulo [81]. The key–column data model can be understood as a multidimensional mapping, mainly including concepts, e.g., *RowKey*, ColumnFamily, and Column. In the key–column data model, a column is the smallest element

stored in the database. It is a triplet of key, value, and timestamp. Any row is organized based on the primary key *RowKey*. In short, the key–column data model simulates the storage format of traditional tables through multilayer mapping. In fact, it is similar to the key–value data model, which needs to be searched by key.

1) Spatio-Temporal Index Based on Cassandra Database: Cassandra is a distributed nonrelational key–value columnar store that can use the gossip protocol to store massive amounts of data on a large number of servers without a single point of failure. Cassandra is a partitioned data store, and data rows are organized based on primary keys. One part is the partition key, which identifies the uniqueness of records in the database and determines where the data are stored. Within a partition, rows are organized together by the rest of the primary keys (e.g., the clustering key or sort key): Cassandra-specific partitioning keys and clustering keys. The cluster key structure can realize multigranularity and multilevel spatio-temporal indexing [82], which effectively promotes the search mechanism.

PSTQ [78] is a Cassandra-based spatio-temporal trajectory query framework. Each row in the track table consists of the partition key Rid, the clustering key $\langle t_{\text{start}}, t_{\text{end}} \rangle$, and the track segment Segmeta. Rid is the unique identifier of the trajectory, and the clustering key ensures that the sequence of trajectories on a node is stored in chronological order. It assigns trajectory segments to Cassandra nodes according to the MurmurHash (Rid) function. PSTQ designs a grid-based secondary index for the spatio-temporal information of the trajectory data, divides the time dimension based on a given time step, and uses the Morton curve to represent the spatial information. For 1-D data, the table partition key consists of three parts:

1) a random number from 0 to 9, which aims to eliminate index hotspots by randomizing the partition fragmentation of each entry;
2) a Z-order value;
3) long time partitions with fixed time step.

Each index entry stores a list of track segment references.

MLS3 [82] is an adaptive spatio-temporal hierarchical indexing method that supports querying of different spatio-temporal feature data. The main ideas are as follows.

1) Spatio-temporal partitioning, which is divided according to time granularity and spatial hierarchy.
2) Information encoding, which jointly encodes spatio-temporal information and stores it in the primary key. The partition key is identified by a larger temporal granularity and a parent spatial grid to ensure that data with spatio-temporal proximity is distributed in the same or adjacent logical partitions. Clustering key is identified by a smaller time granularity and subspace cell.
3) Index is established, and an ML index tree MLS3 with adaptive hierarchical division is proposed.

MLS3 is a global index structure, and the index is stored in Cassandra metadata as an array. The first level is coarse-grained time information, corresponding to the encoding of time information in the partition key; the second level is spatial information, encoded using H-Curve, corresponding to the encoding of spatial information in the partition key; and the third level is fine-grained time information, corresponding to the time information encoding of the cluster key. The first three layers are the initial hierarchical structure, and other layers can be adaptively divided according to the spatio-temporal distribution characteristics of the data.

2) Spatio-Temporal Index Based on HBase Database: The HBase database usually organizes and manages data in the form of key–value pairs. Due to the lack of an effective auxiliary indexing mechanism, it cannot natively support multidimensional queries. For spatio-temporal queries, HBase databases usually perform full table scans on the data, resulting in low efficiency. In recent years, a lot of research work using HBase database to support multidimensional data query has emerged, mainly focusing on the design of rowkey keys, and a small part of supporting data index query in the region. Summarize the spatio-temporal index structure under HBase.

1) Linear index: The spatio-temporal data are converted into one dimension, the Z-order value is used as a rowkey to organize the data, and each region represents an independent spatio-temporal cube region.
2) ML index: The time dimension is used as the first layer of the index, which reduces the length and complexity of the rowkey and improves the retrieval efficiency. The spatial dimension is the second level, and the spatial unit information is represented by linearization technology or tree structure.

UQE-Index [83] is an index framework based on HBase database that supports high insertion throughput and efficient multidimensional query, and builds a local R-tree index that supports efficient retrieval of data within a region. First, divide the data into nonoverlapping time intervals and organize and manage them using B^+-trees; then, use quadtrees to predivide the sampling space under each time interval into multiple subspaces, each subspace is a region, with the data Insert, when the capacity of the subspace exceeds the threshold, the region is split again; second, use the R-tree to organize all subspaces under the management time interval, and add the R-tree to the leaf node of the B^+-tree; finally, establish a local index for each region (R-tree and grid), designed to improve the query performance inside the region. When querying from a region, first get the rowkey by querying the corresponding local index, and then use the rowkey to get the actual data from HBase.

MD-HBase [84] is a multidimensional data management system that supports multidimensional range queries and nearest neighbor queries. The basic idea is: divide the multidimensional space into grids, use the space-filling Z curve to encode the 2-D grid into 1-D data, and take the longest common prefix of the 1-D data as the primary key. MD-HBase only provides spatial indexes in the META surface layer, the time attribute is not considered, so time-related queries and filtering of data in blocks still perform full scan operations, and the query efficiency is low.

STCode [85] adopts a MD index mode, encodes latitude, longitude, and time as Base64-bit string as HBase's *RowKey*. For each dimension, STCode recursively performs binary partitioning, dividing the dimension into two equal parts, with 0 on the left and 1 on the right. Each character in the string

is represented by a 6-bit sequence interleaved in longitude, latitude, and time order, e.g., longitude 1001, latitude 0111, time 1001, bit sequence 101010010111, and Base64 string is fM. Spatio-temporal adjacent points share the STCode prefix. When executing a query, only the *RowKey* with the same prefix as the area needs to be scanned, reducing the delay of search and calculation, reducing the I/O load between client and server.

HBase-STR [86] is an AIS big data index structure built on HBase. *RowKey* consists of five parts:

1) *Geohash:* divide the geographic space into equal subspaces and perform GeoHash encoding, and use R-tree to construct spatial indexes on the subspaces, aiming to accelerate the matching of *RowKey* and GeoHash prefixes;
2) coarse-grained time information;
3) other AIS attributes;
4) fine-grained time information;
5) GeoHash suffix.

The META table establishes a mapping relationship with the RegionServer through the GeoHash prefix. When querying, match the *RowKey* in HBase with the GeoHash prefix to get the data, and it is sorted by year. The date of month and day is filtered to narrow the query scope. Then, the specific ship is found by traversing the MMSI, and the query scope is further narrowed by means of hours, minutes, and seconds. Finally, the GeoHash suffix matches the *RowKey* to obtain the final query result.

HSTI [87] is a multilevel spatio-temporal indexing method based on the HBase database, which is mainly used to support spatio-temporal *k*-NN queries. First divide the entire space into grid cells of equal size and nonoverlapping, and use Z-order technology to map 2-D values into 1-D values Z_n, and Z_n is used as the row key in the META table. Then for each grid cell, the octree is used to divide the spatio-temporal data. When the subspace contains more than the threshold ξ, the subspace is divided. Finally, the Morton code of the octree node is calculated, and the cross layer is performed. Z-order curve filling is designed to quickly retrieve adjacent spaces. Given a spatio-temporal data point p, perform a *k*-NN query q, first computing the spatial Z-order value Z_n, query the regionserver in the META table according to the Z_n value, use the Z-Octree in the RegionServer to retrieve data points, and calculate the adjacent subspace of Z_n and maintain a priority Queue Q, where the priority metric is the distance from the query point q to the point or adjacent subspace, sorted from small to large, and elements in the queue are continuously dequeued until the *k*-NN are found.

SGR-tree [88] is an indexing method that supports queries on moving objects on fixed road networks. First slice in the temporal dimension and divide the slice space into a fine-grained grid. The amount of data in a specific time period T_i is calculated through this grid. Then an R-tree is established for each grid, H-curve coding is performed on the fine-grained grid, and the coding value of the leaf node is judged by the center point of the leaf node, in order to quickly locate the relative position between the MBRs for query.

JUST [89] is a data management engine for processing dynamic trajectory big data. Based on the NoSQL database HBase, the open source project GeoMesa is a spatio-temporal data indexing tool, and Apache Spark is a spatio-temporal data processing tool, which supports spatial range query, spatio-temporal range query, and *k*-NN query. JUST proposes Z2T and XZ2T indexing technologies based on GeoMesa's Z2, XZ2, Z3, and XZ3 spatio-temporal indexes. The Z2T index first divides the time dimension into multiple disjoint time segments and then establishes a separate Z2 index for the data records in each time segment and the Z2T index key. The combination is Num(t)::Z2(lng, lat). The XZ2T index strategy first divides the time dimension into multiple disjoint time periods and then constructs a separate XZ2 index in each time period. JUST supports new data insertion and historical data update without rebuilding the index.

3) Spatio-Temporal Index Based on Accumulo Database: Accumulo supports storing and managing large datasets across clusters. It uses HDFS to store data and uses ZooKeeper for coordination. Accumulo tables exhibit sparsity, ordering, multidimensional mapping and dynamic scalability. As the amount of data becomes larger, Accumulo splits the table divided into smaller parts, called tablets, which can be distributed across multiple table servers. By default, a table will be split into tablets on row boundaries, thus guaranteeing that the entire row is on a single tablet server, and you can set the split point by addingSplits method to control the cutting position of the table. Scanner retrieves a single range of data on a single thread and returns the keys sequentially. BatchScanner uses multiple threads to retrieve multiple ranges of data, which has higher performance, but does not guarantee the return key order ordered. Iterators allow users to implement custom retrievals in the tablet server, so iterators can be used to perform spatio-temporal predicate queries.

DBZ [90] is a globally distributed spatio-temporal index that extends BZ-trees [91], aiming to optimize search performance and space utilization. DBZ supports static indexes and dynamic indexes, and the index mode needs to be specified when creating an index. Static indexes are suitable for initial batch loading of spatio-temporal data. MapReduce jobs are used to scan spatio-temporal data to obtain the spatio-temporal dimension information of each row of ST objects. The spatio-temporal information Z_{key} is cross encoded using the Z3 index [46], [49], and aggregates a certain number of spatio-temporal points in the spatio-temporal interval Z as the leaf nodes of the DBZ tree. The internal nodes are created according to the Zkey value distribution. The dynamic index is maintained by Accumulo, which can adapt to the addition and deletion operations. The data distribution changes to ensure the best search efficiency. When executing a query, extract the time range and space range according to the spatio-temporal query window, convert them into Zkey values through the NoSQL interface layer, and retrieves all servers in the cluster that contain query results by querying index tables.

Fox *et al.* [92] propose a spatio-temporal index based on a sorted distributed storage structure, which aims to query geo-temporal data of spatial geometry composed of single points. The spatial index information is represented as a 35-bit GeoHash, and the time information is encoded as "yyyyMMdd." A part of the GeoHash string is interleaved with part of the datetime string to build the index key. The column family of the

index key contains more fine-grained spatial ranges, e.g., $\langle 01m \rangle$, a finer-grained spatial range encoding, the column qualifier contains the identifier of the data element and a finer-grained spatio-temporal boundary, e.g., $\langle$ident.two0722$\rangle$, where ident. and two represent the data element id and finer-grained spatial range, respectively, and 0722 represents time information dd:hh.

Park *et al.* [93] establish a dynamic multilevel grid index in Accumulo, which can improve the parallelism and adapt to the skew of spatio-temporal data. The main idea is as follows: first use H-Curve technology to map the spatial characteristics of spatio-temporal data stream timestamp, id, location, Record to 1-D data cellID, convert the data into timestamp, id, cellID, location, record. The ingest manager then stores the transformed data in a data buffer consisting of a hash table. The index manager creates a Kd-tree index on the hash table records and stores the index in the index buffer. The two buffer data reach their respective buffer size thresholds Refresh the data table and index table in Accumulo, respectively. Data table key = ID + timestamp, index table key = cellID + timestamp.

GeoMesa [94] is a commercial spatio-temporal database based on distributed systems. It is used for storage, query, and analysis of large-scale spatio-temporal data. It is built on the Apache Accumulo database. The main idea of GeoMesa is to use space-filling curves to reduce multidimensional data into 1-D linear values, and records that are similar in the spatio-temporal dimension are transformed. These data can be efficiently stored and loaded in batches into lexicographically close keys. GeoMesa uses geohashes and timestamps to organize data. A key is a combination of a geohash and a timestamp. It is primarily used for point data Access; line and area data must be broken up into multiple disjoint geohashes. For each indexing strategy, GeoMesa adds a random prefix to the generated key, which distributes the records across region servers and achieves load balancing. When querying data, based on the geohash value, only data items that intersect the query area are considered. GeoMesa uses 3-D Z-order curves to index spatio-temporal point data, and XZ-ordering [95] to index spatio-temporal line/surface data.

So far, for key–column type databases, the research on spatio-temporal indexes has mainly focused on the design of rowkeys. These indexes follow the traditional strategy of generating spatio-temporal joint encoded values to achieve the fastest retrieval speed. But only quickly locate the data block. Physical nodes cannot quickly filter the data in the block.

B. Key–Value Database

The key–value pair data model is actually a mapping. The key is the unique key to find each data address, and the value is the actual stored content of the data. Typically, a hash function is used to realize the mapping from key to value. When querying, the data row is directly located based on the hash value of the key to achieve fast query.

The key–value database that currently supports spatio-temporal data indexing is Redis [96]. Redis is a memory-based database with large data throughput, efficient read and write capabilities, and supports horizontal expansion and high concurrent queries. However, it is mainly aimed at efficient query of single-key values and there are limitations in storage performance, and as the amount of data increases, its storage speed gradually decreases. Therefore, add the "Field" field as the auxiliary filter information of the key. Redis usually provides millisecond-level real-time query, and data persistence tasks are often submitted to other databases, e.g., HBase.

Redis-Geo [86] is an index structure that supports efficient spatio-temporal query of AIS big data. It uses the "Field" field feature of Redis to build a spatial index. When AIS data are written to Redis, the time and MMSI of the AIS data are extracted as the key of the table, the latitude and longitude information is extracted, and the geoadd method is used to add the latitude and longitude to the specified key, and use the entire data as the value. The Redis database can provide millisecond-level real-time response speed, but there are serious limitations in storage performance. When it comes to spatio-temporal big data, it seems stretched.

C. Key–Document Database

MongoDB is a high-performance NoSQL database with built-in support for spatio-temporal indexes [97]. MongoDB uses sharding for horizontal scaling. The user chooses a shard key, which determines how the data in the collection will be distributed. Data are divided into ranges (based on the shard key) and distributed across multiple shards. MongoDB can run on multiple servers, balancing load and/or replicating data to keep systems running in the event of hardware failures.

STIG [23] is an MD index structure of extended Kd-tree, which aims to support complex interactive spatio-temporal range query for static data requiring point-in-polygon (PIP) test. A single index is used to filter spatio-temporal data simultaneously on multiple dimensions to reduce the number of PIP operations. STIG utilizes parallel processors in the GPU to execute multiple independent PIP tests in parallel. STIG tree $k = 2 \times s + m$, where s represents spatial dimension and m represents other attributes, such as time dimension. The index is designed for data with multiple spatio-temporal attributes, such as taxi log data with pickup and drop-off location and time. The tree nodes are divided into internal nodes and leaf nodes, and each leaf node points to a leaf disk block. The internal node of the STIG tree with depth d stores the median of the ($d\%k$) coordinates of the coverage point and pointers to the left and right child nodes. Among them, the leaf node stores pointers to the leaf disk block and k-dimensional borders that define all records in the block. Leaf disk blocks store the property values for which the index was created and pointers to the actual record locations. STIG aggregates points along k dimensions to speedup query processing and maximize utilization of the underlying GPU. STIG does not support dynamic updates, and indexes must be rebuilt periodically when new records are added to the database.

Guan *et al.* [98] propose an ST-Hash for the query processing of spatio-temporal trajectory under frequently updated trajectory data. It first divides the spatio-temporal dimension into half, the range of the child node is equal to half of the parent node interval,

the left node is marked with 0, and the right node is marked with 1, until the desired spatio-temporal granularity is divided. Then, the trajectory point latitude and longitude and time information are mixed and encoded into a 1-D string. For example, (-140, 20, 2015-6-1 00:00:00) is converted to 2015-Re+BP, which shortens the length of the string, reduces the memory space of the index, and improves the index efficiency. The object ID and time information are used as the primary key, and ST-Hash encodes values as new attributes. In addition, B-tree indexes are created on ST-Hash fields to speed up spatio-temporal queries. ST-Hash supports spatio-temporal point queries, spatio-temporal range queries, and spatio-temporal circle queries.

SPTEJ [99] is an MD index structure designed to accelerate the join and query different subjects of the Internet of things under the same spatio-temporal window. The main idea is as follows: divide the spatio-temporal cube into equal-sized cells C_i, $C_i.I$ represents the subject id set that generates tuples in C_i and $C_i.N$ represents the set of subject IDs that generate tuples adjacent cells in C_i. The spatio-temporal join query is divided into the following three steps:

1) find the cells and adjacent cells that contain subjects o_i;
2) retrieve tuples contained in the cell set;
3) sperform join query on candidate tuples.

HBSTR [100] is a hybrid index structure used for spatio-temporal range query of trajectory data and target trajectory search, which consists of spatio-temporal R-tree, hash table, and B^*-tree. The spatio-temporal R-tree is the main structure of the index, which is used to realize the spatio-temporal range query, and the hash table is the auxiliary structure, which is used to store the latest trajectory nodes of all moving objects, which is convenient to find the latest trajectory nodes of the moving objects through the object identifier. B^*-tree is a secondary structure for fast query of target trajectory. The core idea is that the latest trajectory sampling points are centrally managed in groups in the form of trajectory nodes. The trajectory nodes only store the continuous sampling points of a single object, and the trajectory nodes are temporarily stored in the hash table. When a trajectory node is full, it is inserted into the spatio-temporal R-tree as a leaf node, and a new trajectory node is created to receive new sampling points. In addition, a B^*-tree is used to construct a 1-D time index for trajectory nodes to improve query efficiency. HBSTR uses the "weak" requirement feature of MongoDB document structure to store track data and index data in MongoDB independently, thus solving the problem of data storage with unfixed structure.

S4STRD [101] is a scalable in-memory storage system for accessing real-time trajectory data. RAM reserves the temporary storage of field data with high access frequency, and MongoDB is used as auxiliary storage to persist the original data. Index construction processes the following. 1) Spatial index, which divides the geographic space into an $n \times n$ uniform grid, and the coordinates of the grid represent the spatial area location. Merge adjacent low-density trajectory point grids together to form a region, reduce the density gap between grids, realize even distribution of data in space, and establish a one-to-one mapping relationship between regions and servers. 2) Time index, sort the writing time of the data, and establish a B-tree index for the writing time. When a new query is executed, it is searched in RAM according to the index information. When no corresponding result is returned, the original MongoDB data are loaded into the memory according to the index information and the query is executed again.

ST-Index [102] is a spatio-temporal index based on MongoDB database, which supports spatio-temporal range query of massive trajectory data. The core idea is ST-index divides the spatio-temporal cube recursively into $8^{\mathrm{Level}-1}$ subspaces according to the target level *n*. Subspaces (or leaf nodes) are encoded using 3-D Morton [103]. The encoded value ST-code is used as the database key, the value is the trajectory point contained in the subspace, and any trajectory point uniquely belongs to a subspace. When inserting a new trajectory point, calculate the decimal values $\mathrm{Code}_{\mathrm{Lon}}$, $\mathrm{Code}_{\mathrm{Lat}}$ and $\mathrm{Code}_{\mathrm{Time}}$ of the longitude, latitude and time of the trajectory point, and then convert them into n-bit binary sequences Lat_B, Lon_B and Time_B, respectively. Interleave the bits of the three binary sequences and convert them into Morton codes. According to the Morton code value, the corresponding key value is found in the database. When ST-index performs a spatio-temporal range query, the spatio-temporal query is converted into a 1-D query of ST-code.

The "weak" requirement feature of MongoDB document structure can flexibly handle data of different structures, the automatic sharding mechanism can achieve load balancing, support the creation of single and composite indexes on specified attributes, and can combine with external data structures to build spatio-temporal indexes, but its storage mode mainly suitable for data models with embedded or reference relationships; therefore, it is difficult to efficiently process spatio-temporal data.

D. Summary

This article mainly discusses spatio-temporal indexes based on NoSQL databases. As with all databases, fast access to raw data in NoSQL databases depends on the structural organization of the stored information and the availability of suitable indexing methods. Well-designed data structures facilitate the use of simple techniques quickly extract the required information from a collection of data, and sophisticated indexing methods can be used to quickly locate single or multiple objects in a database. However, many queries tend to support some specific application.

The flexible storage mode of the NoSQL database provides simplicity for the storage and retrieval of spatio-temporal big data, but the strong encapsulation of the system and the storage structure of the fixed mode make the spatio-temporal index research only focus on the design of the *RowKey*. Therefore, the spatio-temporal index based on the NoSQL database has three development trends, which are summarized as follows.

1) Multi-level index mode: Mature linear indexing technology has gradually become the main means of managing spatio-temporal data in NoSQL databases. However, sufficient spatio-temporal query operations require index delinearization, which tends to develop in a ML index mode.

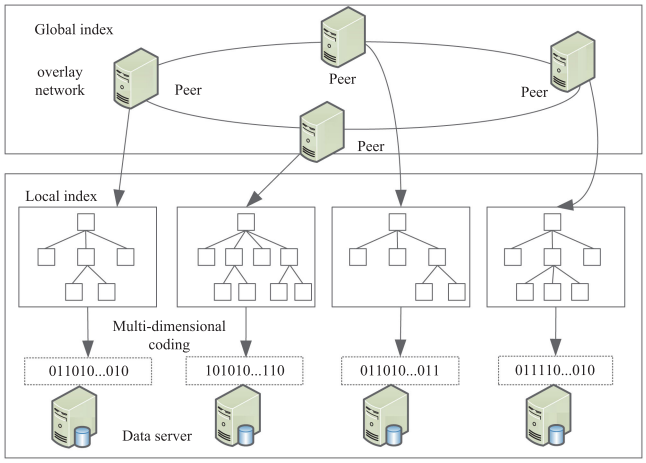

Fig. 9. MD indexing in standalone distributed system.

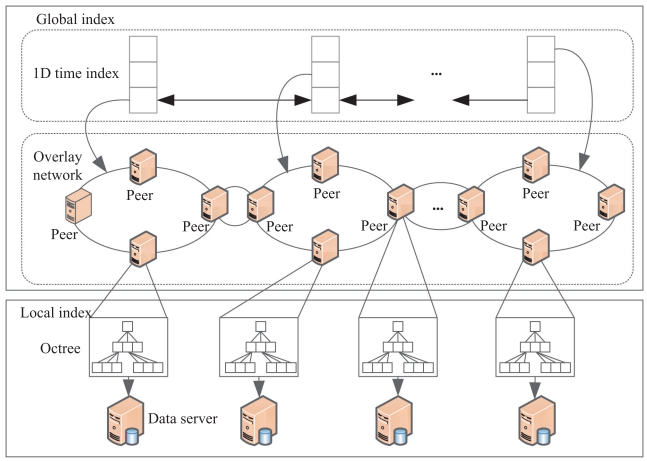

Fig. 10. ML indexing in standalone distributed system.

2) Single index platform: Redis low scalability and MongoDB "loose" data structure make key–column database a mainstream NoSQL database for spatio-temporal big data management.
3) Diversified index structure: The existing rowkey mainly supports the global index, and the research on supporting the local index of region data is imminent.

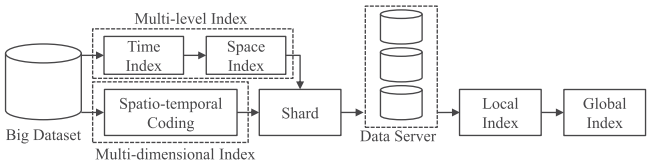

Fig. 11. General index flowchart over standalone parallel and distributed systems.

IV. Spatio-Temporal Indexing for Standalone Parallel and Distributed Systems

There are also some researches devoted to building spatio-temporal indexes on an independent and scalable distributed system for efficiently querying target data from spatio-temporal big data. Such spatio-temporal indexing methods do not depend on existing distributed computing systems and NoSQL databases, but are built on a distributed system using a P2P node architecture. The advantages are as follows.

1) Consistent hashing is used to distribute data, which avoids the data skew problem caused by spatio-temporal indexes.
2) Node decentralization is a good way to avoid the single point of failure problem in the master–slave architecture.

A. Index Mechanism

At present, the spatio-temporal indexes in independent parallel and distributed systems mainly adopt two mechanisms: ML index and MD index. The general index flowchart over standalone parallel and distributed systems is shown in Fig. 11.

1) Multidimensional Spatio-Temporal Indexing Schema Under Standalone Distribution: As shown in Fig. 9, first, the spatio-temporal dimension information is jointly encoded, and the sharding mechanism is used to store the codes with the same features in the same data server. Then, in order to speedup the matching speed of encoding, a local index based on B-tree or B^+-tree is constructed. Finally, the local index of each data server is published to the peer nodes in the overlay network to form a global index, and each local index uniquely corresponds to a node. For each data server in the overlay network, it plays two roles. The data storage node is also an index node. At the same time, as an index node, the server node is both a global index node and a local index node. 1) As a global index node, the server undertakes the tasks of global routing and positioning and provides the interface between the global index and the local index. 2) As a local index node, the data server is responsible for providing the index of local data and publishes data information to the global index layer as needed.

Query execution process: Randomly send the query request proposed by the user to any node in the network, go to the node to find out whether there are data that satisfy the query, if so, perform a specific data retrieval operation and return a local result set. Then, forward the query request to the next node through routing and perform the same steps; if not, skip the node and directly forward the query request to the next node.

2) Multilevel Spatio-Temporal Index Mode Under Standalone Distribution: As shown in Fig. 10, the time information is firsts divided into small-granularity time ranges, and indexes are established to speed up the matching of time information. For each time range, a distributed cluster of P2P architecture is constructed, and the data are distributed to the node server through the hash function. The overlay network stores the routing information and data information of adjacent nodes with each other to achieve cross-network communication and data retrieval. Then, a local index is built for the local data on each node server. Each node server in the overlay network acts as two role, it is both a data storage node and a local index node.

Query execution process: Extract the time range of the user's query request, find the corresponding overlay network through

a 1-D time index, send the query to any node in the network, and execute. 1) Determine whether there are data satisfying the query under the node search, if there is, perform a specific data retrieval operation, return the local result set, and then forward the query request to the next node through routing, and perform the same steps; if not, skip the node and directly forward the query request to the next node. 2) Judge whether the adjacent nodes of this node have data that satisfy the query, if so, perform a cross-network data retrieval operation, return the adjacent node result set, and then forward the query request to the next node through routing, and perform the same steps; if not, skip the adjacent node and directly forward the query request to the next node.

B. Individual Prototype and Techniques

Elite [2] is a method that supports parallel update and query processing of spatio-temporal big data, and supports spatio-temporal range query (STRQ) and spatio-temporal nearest neighbor query (STNNQ). The Elite index consists of three layers: 1) skip list layer, 2) ring layer, and 3) OCT tree layer. The skip list layer and ring layer constitute a global index, and the OCT tree layer is a local index and contains an OCT tree and hash table, the OCT tree stores the trajectory observation positions, and the hash table maps the unique identifier ID of the trajectory to the earliest observation position and the latest observation position of the trajectory. The skip list layer contains a doubly linked skip list, where each node in the skip list corresponds to a torus cluster. A node's key consists of the time interval of the ring cluster and a pre-allocated segment of consecutive IP addresses. The ring layer consists of chain rings, in which each ring body is a cluster of nodes. Each node in the ring maintains a routing table that contains adjacent node IP addresses and data ranges. The information in the routing table is used for communication between nodes within the ring. For communication between two rings, a ring node randomly selects an IP address from its IP address segment connecting the rings, and then, the randomly selected node communicates within the ring to find the target node. The trajectory query is divided into three stages.

1) *Filter:* Identify ring nodes that overlap the query area with a distributed index. All candidate circle nodes run the corresponding subqueries in parallel. Access the local index to retrieve candidate trajectories.
2) *Node assignment:* Assign free circle nodes to each candidate node to perform further result set refinement.
3) *Refinement:* Use a sampling method to generate a set of possible instances, simulate the uncertainty of the trajectory, calculate the corresponding qualified probability, and then combine the query results of all nodes to form the final result.

GeoTrie [104] is a MD index structure based on P2P mode, which supports efficient spatio-temporal range queries on distributed hash tables (document PHT [105] addresses the support of range queries on distributed hash tables). It can add or delete nodes arbitrarily without destroying the data index structure, and avoid load balancing problems caused by large data storage and concurrent queries. The following are the solution steps.

1) *Mapping:* Convert the latitude, longitude and time to 32-bit binary strings respectively, forming a tuple $T_k = (T_{\text{lat}}, T_{\text{lon}}, T_t)$.
2) *Index:* Map all GeoTrie nodes to the DHT structure through the $K = \text{Hash}(l)$ method.

The spatio-temporal data within the node range are stored according to the hash consistency principle. When executing the query task, convert the spatio-temporal window into a 32-bit binary string, calculate the minimum common prefix label for each binary string (the length of the minimum common prefix label remains the same), and directly query in the octree according to the minimum common prefix label. There is no need to start the query from the root node of the tree, which shortens the query time.

A-tree [106] solves the distributed indexing problem of multidimensional data in cloud computing environment. The basic idea is: the master node builds a global index through a 1-D array, and each index on the array is assigned to a node. Each slave node builds an R-tree and creates a Bloom filter at the same time. The data are allocated according to the DHT principle to the node storage. When querying a point, first verify it through Bloom filter. If the query point is not in it, then do not perform the R-tree query, otherwise continue to perform the R-tree query. When querying a range, you cannot use the Bloom filter, you must do an R-tree query.

Galileo [107] is a high-throughput storage system using zero-hop DHT peer-to-peer network architecture, supporting hundreds of millisecond read and write operations. A single storage node in the system is divided into multiple groups, and each group is assigned a unique UUID stream. Galileo designs a two-layer hashing scheme: first, the GeoHash is calculated from the data space information to determine the target group of the data. Then, use temporal and characteristic metadata sets of data compute SHA-1 hashes to determine storage nodes within a group. Group and storage node hashes can determine the specific node UUIDs with which to communicate. Galileo utilizes the host file system to store data on physical media, the unit of storage is called a block, and each block carries a set of metadata, which contains both temporal and spatial information. When queried, it incrementally returns the metadata of matching blocks to the requester and organizes these metadata blocks into traversable in-memory metadata set subgraph. Galileo can freely switch the root node of the dataset subgraph traversal through redirection to meet the needs of different spatio-temporal application queries.

DISTIL+ [108] is a distributed spatio-temporal data processing system for highly updated location data that extends DISTIL [109]. DISTIL+ builds a three-tier distributed in-memory index by leveraging the asynchronous partitioned global address space paradigm, aiming to support highly concurrent spatio-temporal range queries and spatio-temporal *k*-NN queries. The location, velocity, orientation, and timestamp records of moving objects are stored in the location table. The main idea of index design is to discretize the spatial and temporal dimensions. The global index (level 1) adopts a spatial domain-based quadtree partition. The local index (levels 2 and 3) of each node consists of a spatial index and a partial temporal index (PTIndex). The

spatial index maintains information about each grid cell in the spatial domain, and each grid cell has a PTIndex, which is defined by an interval Table composition, each entry in the interval table contains a bit vector (Bit-vector) and a hash table (RIDListMap), the former identifies the moving objects in a specific grid cell within a given time interval, RIDListMap will each A moving object is associated with a list of record identifiers that locates the actual moving record of the moving object in the location table. When a new location update is received from the moving object, the location record object LRec is inserted into the concurrent queue by the coordinator component, and then use the producer–consumer paradigm to implement parallel processing.

Toss-it [110] supports indexing the current position of the moving object. The main idea is to build a new index every time the location changes, instead of updating the old one, so that there is no need to maintain a centralized update buffer to maintain indexes that have not been updated yet, improving the scalability of the system. ToSS-it uses Voronoi diagrams distributed over multiple nodes, first distributing all objects on cloud servers while maintaining their spatial locality, constructing Voronoi diagrams in a first-distribution-then-build fashion. The initial distribution of data is executed using a centralized server. Then, a local Voronoi diagram (LVD) is built on each server, and the LVD decomposes the space into disjoint polygons. The generation of the LVD utilizes all available cores of the CPU to further expand on each node and divide objects. Build a hierarchical Voronoi index structure on each server to speedup query processing. Submit a query q to a node N_q, then nodes IN_q that intersect the query area will be found, and the query will be forwarded to these nodes, Queries are run in parallel in INq nodes using LVD, and partial results of the query on each node are sent back to Nq for aggregation. D-ToSS [111] is an enhanced version of ToSS-it, in which D-ToSS does not partition data across nodes and a centralized server is required.

The spatio-temporal indexes discussed previously use a variety of indexing techniques to improve spatio-temporal data processing capabilities without relying on existing distributed computing systems and NoSQL databases. These indexes satisfy scalability and fault tolerance, but the disadvantages are as follows: 1) Consistent hashing is used to distribute data, although load balancing is achieved, but it weakens the spatio-temporal relationship between data and destroys data locality, resulting in indexes that can only satisfy simple spatio-temporal data point queries, but cannot meet the needs of more complex spatio-temporal applications; 2) the decentralization of nodes makes it difficult to directly address the index server whether it is an ML index or an MD index. When data are inserted frequently, index updates take a long time.

V. Comparison of Indexing Mechanisms

In this section, we summarize three main indexing mechanisms and indexing methods from three distributed environments, and compare and analyze the advantages and disadvantages of these indexing mechanisms from four aspects: adaptive, data skew, index size, and query efficiency. Finally, the typical application scenarios of each indexing mechanism are given in Table III.

In terms of adaptive, this metric can be described as finding the best match for a given query workload and choosing when the system should adjust its storage at runtime. Both ML idnex and Joint index is able to identify changes in query workload.

In dealing with data skew, this metric aims to determine the most appropriate partitioning technique for a given dataset in the presence of data skew. ML index select the most suitable partitioning technology for each level and the partitioning technology between levels may be different. Therefore, ML index solve this problem. MD index and joint index typically split the spatiotemporal dimension into uniform cells and are, therefore, not suitable for dealing with skewed data.

In terms of query efficiency, ML index has a higher query efficiency when dealing with range query, *k*-NN query, and similarity query. When processing join queries, the data of each layer need to be processed, so the efficiency is moderate. MD index can handle fast processing of single-dimensional and multidimensional queries, ignoring the effect of partition order on the nature of partitions, so MD index can efficiently handle range queries. The joint index can accurately and quickly find the location of the data through the key value, so it has high efficiency in processing conditional queries, such as range query, *k*-NN query, and similarity query.

To sum up, ML indexes are suitable for queries focused on the time or space dimension, built on distributed computing systems and independent parallel and distributed systems, with custom index granularity and high throughput requirements. MD indexes are suitable for processing data evenly distributed, general range queries, built on distributed computing systems and independent parallel and distributed systems, with high throughput, high real-time, and low latency requirements. Joint index is suitable for building NoSQL databases, with data update, data insertion, high real-time, and low latency requirements.

VI. Experimental Results

In this section, we compare and analyze the performance of spatio-temporal indexing over different distributed systems.

A. Settings

To evaluate the performance of indexing methods, we use a real spatio-temporal datasets: T-Drive,[1] which contains a one-week trajectories of 10 357 taxis. The total number of points in this dataset is about 15 million. Table IV lists the softwares and their versions used in our experiments.

B. Spatio-Temporal Index Performance Over Batch System

In the performance comparison of spatio-temporal indexing over batch system, we choose ST-Hadoop,[2] HT, STQuery, and CloST[3] with open-source code or clear algorithm flow as comparison methods, and choose the query time varying with the

[1] [Online]. Available: https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/

[2] [Online]. Available: https://github.com/lmarabi/st-hadoop

[3] [Online]. Available: https://github.com/douglasapeixoto/spark-trajectory-storage

TABLE III
COMPARISON OF DIFFERENT INDEXING MECHANISMS

Index Mechanisms	Index Methods	Literatures	Adaptive[c]	Data skew[d]	Index Size	Query Efficiency	Application Scenarios
ML index[a]	Global/local index: time index, space partition(e.g., R-tree, Kd-tree, Quadtree, Grid)/R-tree Global index: time index, space partition or time index, space partition, time index or space index, time index	[38]–[40], [50], [64], [67], [72], [101]	✓	✓	Large	Range query, *k*-NN query, similarity query: high Join query: medium	Queries focus on the temporal or the spatial dimension, distributed computing system, standalone parallel and distributed system, custom index granularity, high throughput
MD index[b]	Global/local index: Octree, 3DR-tree, equal-sized cube/R-tree Global index in the form of Octree, 3DR-tree	[23], [39], [41], [99], [104]	–	–	Medium	Range query: high Join query, *k*-NN query, similarity query: medium	Data uniform distribute and general range query, distributed computing system, standalone parallel and distributed system, high throughput, high real-time, low latency
Joint index	The space-time space is first divided into equal-sized cubes using an octree, and then a linear encoding technique is used for all cubes. For each dimension, it recursively performs binary partitioning, dividing the dimension into two equal parts, with 0 on the left and 1 on the right	[78], [85], [87], [89], [94], [98], [102]	✓	–	Small	Range query, *k*-NN query, similarity query: high Join query: low	NoSQL database, data update, data insert, high real-time, low latency

[a]ML index. [b]MD index. [c]Aims to find the best match for a given query task. [d]In the case of skewed data, identify the most appropriate partition technique for the dataset.

TABLE IV
SOFTWARES IN THE EXPERIMENTS

Software	Version	Software	Version	Software	Version
Hadoop	2.7.2	Storm	2.3.0	Spark	2.3.3
HBase	1.4.9	JDK	1.8	Scala	2.1.1

size of the data, query time, index time, and index size as the evaluation indicators for comparison.

We compare the performance of the indexing method for spatio-temporal range queries with different data sizes, where the spatio-temporal window remains unchanged by default. As shown in Fig. 12(a), ST-Hadoop has the fastest query speed, because 1) ST-Hadoop adopts an ML indexing mechanism, which can make better use of machine hardware resources, and 2) the global/local index mode of ST-Hadoop can quickly locate the location of the physical node where the data block is located.

Fig. 12(b) shows the changes in the performance of the spatio-temporal range query under different spatio-temporal query windows. We randomly set 100 different query windows and each query is executed only once, since the cost of different queries can vary widely, we report query times using 5%–95% intervals and the median of these 100 queries. As shown in Fig. 12(b), ST-Hadoop has led to the best query time among all index methods. Although CloST also adopts an ML indexing mechanism, its query performance is lower than ST-Hadoop. The main reason is that CloST's indexing method cannot handle skewed data, which can easily lead to load imbalance.

As shown in Fig. 12(c), the storage size under different spatiotemporal indexes is compared with the change of data volume. The horizontal axis represents the percentage of usage data in the corresponding dataset. Note that the storage index size here includes the data itself. We can know from the figure that

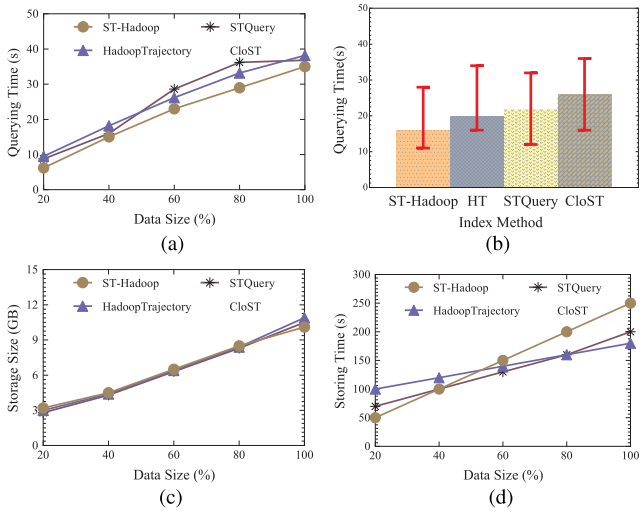

Fig. 12. Performance comparison of different indexing methods over the Hadoop framework. (a) Query time over data size. (b) Query time over index method. (c) Index size. (d) Index time.

1) as the amount of data increases, the size of all spatiotemporal indexes increases linearly, and 2) the ST-Hadoop ML index is larger than the HT index size, because the ML index may copy data once at each level.

Fig. 12(d) shows how the storage index time varies with the size of the data. It can be seen from the figure that with the increase of data volume, for all indexing methods, the storage indexing time increases exponentially. ST-Hadoop takes more time than HT, the main reason is that ST-Hadoop needs to process data hierarchically, under Hadoop architecture, this will generate more intermediate result storage, which means more IO operations are required.

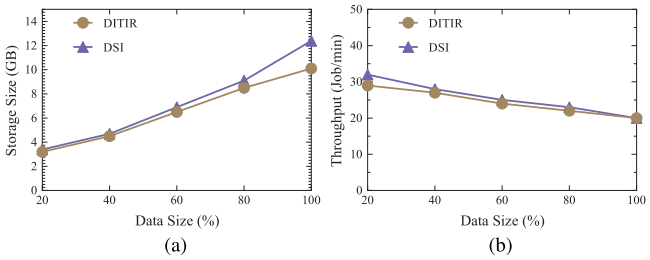

Fig. 13. Performance comparison of different indexing methods over the stream processing system. (a) Index size. (b) Throughput.

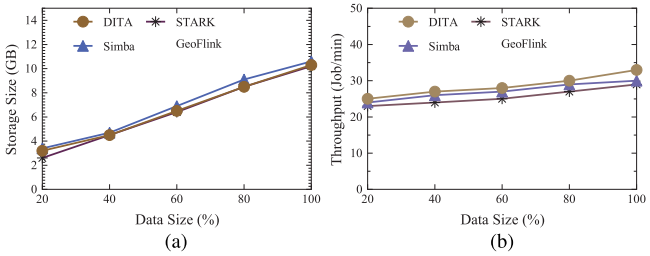

Fig. 14. Performance comparison of different indexing methods over the hybrid processing system. (a) Index size. (b) Throughput.

C. Spatio-Temporal Index Performance Over Stream Processing System

In the performance comparison of spatiotemporal indexing under batch system, we choose DITIR and DSI with clear algorithm flow as comparison methods, and choose index size and throughput as the evaluation indicators for comparison.

Fig. 13(a) shows the storage cost of the two methods at different raw data sizes. Note that the storage cost includes the index structure and the data. As we can see, with more raw data, it requires more storage space to store both datasets. Clearly, DSI takes up more space than DITIR, mainly because DSI build indexes in the x-axis and y-axis directions, respectively.

Fig. 13(b) shows the throughput of the two methods at different raw data sizes. As we can see, as the amount of data increases, the throughput of both methods decreases, mainly because the two indexing methods do not consider load balancing.

D. Spatio-Temporal Index Performance Over Hybrid Processing System

In the performance comparison of spatiotemporal indexing over batch system, we choose DITA,[4] Simba,[5] STARK,[6] and GeoFlink[7] with open-source code or clear algorithm flow as comparison methods, and choose index size and throughput as the evaluation indicators for comparison.

Fig. 14(a) shows the storage cost of the four methods at different raw data sizes. Note that the storage cost includes the index structure and the data. As we can see, with more raw data, it requires more storage space to store both datasets. For the dataset, DITA and Simba takes up more space than other methods, mainly because they use a ML indexing mechanism to deal with the spatio-temporal dimension separately.

Fig. 14(b) shows the throughput of the four methods at different raw data sizes. As we can see, although DITA and Simba take up more space than other methods, their throughput is higher, and the global/local index mode can make full use of machine hardware resources and improve query performance.

[4][Online]. Available: https://github.com/TsinghuaDatabaseGroup/DITA
[5][Online]. Available: https://github.com/InitialDLab/Simba
[6][Online]. Available: https://github.com/dbis-ilm/stark
[7][Online]. Available: https://github.com/aistairc/SpatialFlink

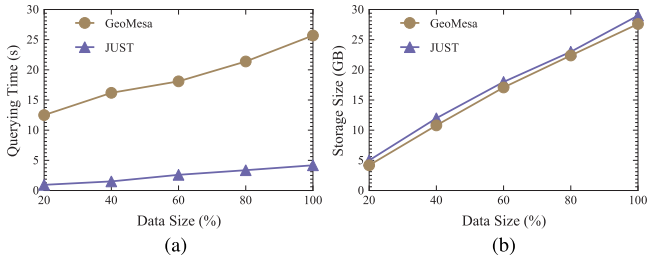

Fig. 15. Performance comparison of different indexing methods over the HBase database. (a) Query time. (b) Index size.

E. Spatio-Temporal Index Performance Over HBase Database

In the performance comparison of spatio-temporal indexing over HBase database, we choose GeoMesa[8] and JUST[9] with open-source code as comparison methods, and choose query time and index size as the evaluation indicators for comparison.

As shown in Fig. 15(a), as the dataset is larger, both methods require more time to answer the spatiotemporal range query because more data are scanned and returned. JUST is much faster than GeoMesa because JUST encodes both temporal and spatial information into the keys of the NoSQL data store, and encodes temporal information first and spatial information second, which allows us to quickly locate eligible records directly.

Fig. 15(b) shows the storage cost of the two methods at different raw data sizes. Note that the storage cost includes the index structure (i.e., keys) and the data (i.e., values) itself. As we can see, with more raw data, it requires more storage space to store both datasets. JUST takes up more space than GeoMesa, mainly because JUST applies more space to store index information.

VII. Challenges and Outlook

Most of the existing prototypes and systems adopt distributed indexing, and the basic idea is to adopt a two-level indexing scheme. At the local indexing layer, indexing structures, e.g., R-tree, Quadtree, and grid are usually used. Another way is to use spatial fill curves map data into 1-D values and use traditional B-tree for local indexing. NoSQL databases widely adopt the latter approach. In the case of spatio-temporal data, some indexing methods consider the temporal dimension first, then the spatial dimension. In the global index layer, the most common method is

[8][Online]. Available: https://github.com/locationtech/geomesa
[9][Online]. Available: http://just.urban-computing.com/code/compare_platform.rar

to collect summary information from the local indexes of nodes in order to build a global index and quickly locate queries to data nodes. The index also brings new challenges and future research directions.

A. Multimodal Adaptive Indexing Mechanism for Spatio-Temporal big data in Multicopy Mode

In the conventional big data storage environment, multiple copies are the basic strategy to ensure data security and reliability. A unified index mode cannot support the diverse application requirements of spatio-temporal big data, and different index modes are established for different copies of the same data, it can improve the storage access efficiency of data according to different application requirements. Therefore, how to design a multi-modal adaptive spatio-temporal big data indexing mechanism is a hot research direction.

B. Spatio-Temporal Index Mode for Low-Latency Queries in High Real-Time Applications

With the application of 5G network, in real-time application fields, e.g., unmanned driving, real-time navigation, and short-term prediction, the requirements for retrieval efficiency of spatio-temporal big data is getting higher and higher. Considering the communication cost, maintenance cost, and index performance, neither the tree-based index nor the grid index can be directly applied to the streaming data processing system. Therefore, how to design a spatio-temporal index mode with low communication cost, low maintenance cost, and high index performance on the streaming data processing system, which can shorten the query response time of petabyte-level data to the second level, has received more and more attention.

C. Spatio-Temporal Indexing Technology and Theoretical Specification Based on NoSQL Database

At present, the existing spatio-temporal indexes based on NoSQL databases are only oriented to specific applications, the supported queries, and key technologies used are very different, and a set of systematic normative criteria has not been formed. Therefore, it is urgent to standardize NoSQL spatio-temporal indexes and their support theories, which mainly include the following.

1) Specification technology and supported query operations for NoSQL spatio-temporal index.
2) The guidelines for application-oriented design of NoSQL spatio-temporal indexes, the spatio-temporal index structure for minimizing spatio-temporal granularity, and the heuristic criteria for minimizing query response time provide guidelines for the optimal spatio-temporal index structure.
3) Study the spatio-temporal semantics expressed by the spatio-temporal index and the correctness verification rules to provide a certain basis for the rationality and correctness of the spatio-temporal data organization.

VIII. Conclusion

Spatio-temporal indexing is the core content of spatio-temporal big data management, especially with the introduction of concepts, e.g., knowledge computing, social computing, and urban computing [112]. Various parallel and distributed spatio-temporal indexing methods to improve access efficiency to spatio-temporal big data. This review covers spatio-temporal indexing methods proposed from 2010 to 2020, namely:

1) spatio-temporal indexing based on distributed computing system;
2) spatio-temporal indexing based on NoSQL database;
3) spatio-temporal indexing for standalone parallel and distribution systems.

Acknowledgment

The authors would like to thank to X. Du, C. Li, G. Peng, and C. Li, with special thank A. Eldawy, Assistant Professor, with the University of California Riverside. This article was produced by the IEEE Publication Technology Group.

References

[1] D. Yao, C. Zhang, J. Huang, Y. Chen, and J. Bi, "Semantic understanding of spatio-temporal data: Technology & application," *J. Softw.*, vol. 29, no. 7, pp. 2018–2045, Apr. 2018.
[2] X. Xie, B. Mei, J. Chen, X. Du, and C. Jensen, "Elite: An elastic infrastructure for big spatio-temporal trajectories," *Very Large Data Bases J.*, vol. 25, no. 4, pp. 473–493, Aug. 2016.
[3] X. Zhou, D. Qin, L. Chen, and Y. Zhang, "Real-time context-aware social media recommendation," *Very Large Data Bases J.*, vol. 28, no. 2, pp. 197–219, Apr. 2019.
[4] K. Huang, G. Li, and J. Wang, "Rapid retrieval strategy for massive remote sensing metadata based on GeoHash coding," *Remote Sens. Lett.*, vol. 9, no. 11, pp. 1070–1078, Nov. 2018.
[5] J. Xia, C. Yang, and Q. Li, "Building a spatio-temporal index for Earth observation Big Data," *Int. J. Appl. Earth Observ. Geoinf.*, vol. 73, pp. 245–252, Dec. 2018.
[6] Z. Li *et al.*, "A spatio-temporal indexing approach for efficient processing of big array-based climate data with MapReduce," *Int. J. Geographical Inf. Sci.*, vol. 31, no. 1, pp. 17–35, Jan. 2016.
[7] M. Zhu, C. Liu, and Y. Han, "Approach to discovering companion patterns based on traffic data stream," *IET Intell. Transp. Syst.*, vol. 12, no. 10, pp. 1351–1359, Dec. 2018.
[8] A. Mahmood, S. Punni, and W. Aref, "Spatio-temporal access methods: A survey (2010-2017)," *Geoinformatica*, vol. 23, no. 1, pp. 1–36, Jan. 2019.
[9] K. Jitkajornwanich, N. Pant, M. Fouladgar, and R. Elmasri, "A survey on spatial, temporal, and spatio-temporal database research and an original example of relevant applications using SQL ecosystem and deep learning," *J. Inf. Telecommun.*, vol. 4, no. 4, pp. 524–559, Sep. 2020.
[10] T. Zäschke, C. Zimmerli, and M. Norrie, "The PH-tree: A space-efficient storage structure and multi-dimensional index," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2014, pp. 397–408.
[11] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2119–2133.
[12] H. Wang, K. Zheng, X. Zhou, and S. Sadiq, "SharkDB: An in-memory storage system for massive trajectory data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1099–1104.
[13] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," in *Proc. Int. Conf. Data Eng.*, 2010, pp. 109–120.
[14] S. Ray, "Toward high performance spatio-temporal data management systems," in *Proc. IEEE 15th Int. Conf. Mobile Data Manage.*, 2014, pp. 19–22.
[15] E. Carneiro, A. Carvalho, and M. Oliveira, "I2B+ tree: Interval B+-tree variant toward fast indexing of time-dependent data," in *Proc. 15th Iberian Conf. Inf. Syst. Technol.*, 2020, pp. 1–7.

[16] D. That, I. Popa, and K. Zeitouni, "TRIFL: A generic trajectory index for flash storage," *ACM Trans. Spatial Algorithms Syst.*, vol. 1, no. 2, pp. 1–44, Jul. 2015.

[17] A. Mahmoo, A. Aly, T. Kuznetsova, S. Basalamah, and A. WG, "Disk-based indexing of recent trajectories," *ACM Trans. Spatial Algorithms Syst.*, vol. 4, no. 3, pp. 1–27, 2018.

[18] M. Singh, Q. Zhu, and H. Jagadish, "SWST: A disk based index for sliding window spatio-temporal data," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 342–353.

[19] A. Mahmood, W. Aref, A. Aly, and S. Basalamah, "Indexing recent trajectories of moving objects," in *Proc. SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2014, pp. 393–396.

[20] T. Nguyen, Z. He, and Y. Chen, "SeTPR*-tree: Efficient buffering for spatiotemporal indexes via shared execution," *Comput. J.*, vol. 56, no. 1, pp. 115–137, 2013.

[21] D. šidlauskas, K. Ross, C. Jensen, and S. Šaltenis, "Thread-level parallel indexing of update intensive moving-object workloads," in *Proc. Int. Symp. Spatial Temporal Databases*, 2011, pp. 186–204.

[22] D. Šidlauskas, S. Šaltenis, and C. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 37–48.

[23] H. Doraiswamy, H. Vo, C. Silva, and J. Freire, "A GPU-based index to support interactive spatio-temporal queries over historical data," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 1086–1097.

[24] W. Tang, W. Feng, and M. Jia, "Massively parallel spatial point pattern analysis: Ripley's K function accelerated using graphics processing units," *Int. J. Geographical Inf. Sci.*, vol. 29, no. 3, pp. 412–439, Mar. 2015.

[25] B. Zheng, J. Xu, W.-C. Lee, and L. Lee, "Grid-partition index: A hybrid method for nearest-neighbor queries in wireless location-based services," *Very Large Data Bases J.*, vol. 15, no. 1, pp. 21–39, Jan. 2006.

[26] Z. Chen, L. Chen, G. Cong, and C. S. Jensen, "Location- and keyword-based querying of geo-textual data: A survey," *Very Large Data Bases J.*, vol. 30, no. 4, pp. 603–640, Mar. 2021.

[27] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984. pp. 47–57.

[28] Geohash, 2014. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[29] K. Lee, W. Lee, B. Zheng, H. Li, and Y. Tian, "Z-SKY: An efficient skyline query processing framework based on Z-order," *Very Large Data Bases J.*, vol. 19, no. 3, pp. 333–362, Jun. 2010.

[30] C. Düntgen, T. Behr, and R. Güting, "BerlinMOD: A benchmark for moving object databases," *Very Large Data Bases J.*, vol. 18, no. 6, pp. 1335–1368, Apr. 2009.

[31] W. Guo, Y. Zhao, G. Wang, and L. Wei, "Efficient fault-tolerant processing technology for Flink iterative computing," *Chin. J. Comput.*, vol. 43, no. 11, pp. 2101–2118, Nov. 2020.

[32] Apache Hadoop, 2011. [Online]. Available: https://hadoop.apache.org

[33] Apache Storm, 2017. [Online]. Available: http://storm.apache.org

[34] S4 Apache, 2011. [Online]. Available: https://github.com/apache/incubator-retired-s4

[35] Apache Spark, 2012. [Online]. Available: http://spark.apache.org

[36] Apache Flink, 2015. [Online]. Available: http://flink.apache.org

[37] A. Zhou, B. Yang, Z. Jin, and Q. MA, "Location-based services: Architecture and progress," *Chin. J. Comput.*, vol. 34, no. 7, pp. 1155–1171, Jul. 2011.

[38] L. Alarabi and M. Mokbel, "A demonstration of ST-hadoop: A mapreduce framework for big spatio-temporal data," in *Proc. Very Large Data Bases Endow.*, 2017, pp. 1961–1964.

[39] L. Han, L. Huang, X. Yang, W. Pan, and K. Wang, "A novel spatio-temporal data storage and index method for ARM-based hadoop server," in *Proc. Int. Conf. Cloud Comput. Secur.*, 2016, pp. 206–216.

[40] H. Tan, W. Luo, and L. Ni, "CloST: A hadoop-based storage system for big spatio-temporal data analytics," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2012, pp. 2139–2143.

[41] M. Bakli, M. Sakr, and T. Soliman, "HadoopTrajectory: A hadoop spatio-temporal data processing extension," *J. Geographical Syst.*, vol. 21, pp. 211–235, Jun. 2019.

[42] J. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[43] S. Leutenegger, M. Lopez, and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *Proc. 13th Int. Conf. Data Eng.*, 1997, pp. 97–506.

[44] R. Schneider, N. Beckmann, H. P. Kriegel, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1990, pp. 322–331.

[45] Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: An optimizedspatio-temporal access method for predictive queries," in *Proc. Very Large Data Bases Endowment*., 2003, pp. 790–801.

[46] Y. Theodoridis, M. Vazirgianni, and T. Sellis, "Spatio-temporal indexing for large multimedia applications," in *Proc. IEEE Int. Conf. Multimedia Comput. Syst.*, 1996, pp. 441–448.

[47] A. Eldawy, L. Alarabi, and M. Mokbel, "Spatial partitioning techniques in spatial-hadoop," in *Proc. Very Large Data Bases Endow.*, 2015, pp. 1602–1605.

[48] A. Eldawy and M. Mokbel, "SpatialHadoop: A MapReduce framework for spatial data," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1352–1363.

[49] C. Jackins and S. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Comput. Graph. Image Process.*, vol. 14, no. 3, pp. 249–270, Feb. 1980.

[50] D. Rammer, S. Pallickara, and S. Pallickara, "ATLAS: A distributed file system for spatiotemporal data," in *Proc. 12th IEEE/ACM Int. Conf. Utility Cloud Comput.*, 2019, pp. 11–20.

[51] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2013, pp. 38–49.

[52] D. Sun, G. Zhang, and W. Zheng, "Big Data stream computing: Technologies and instances," *J. Softw.*, vol. 25, no. 4, pp. 839–862, Jan. 2014.

[53] R. Cai, Z. Lu, L. Wang, Z. Zhang, T. Fur, and M. Winslett, "DITIR: Distributed index for high throughput trajectory insertion and real-time temporal range query," in *Proc. Very Large Data Bases Endowment*., 2017, pp. 1865–1868.

[54] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," *IBM Ltd.*, Ottawa, 1966.

[55] P. Mazumdar, L. Wang, M. Winslet, Z. Zhang, and D. Jung, "An index scheme for fast data stream to distributed append-only store," in *Proc. Int. Workshop Web Databases*, 2016, pp. 1–6.

[56] F. Zhang *et al.*, "Real-time spatial queries for moving objects using storm topology," *ISPRS Int. J. Geo-Inf.*, vol. 5, no. 10, Sep. 2016, Art. no. 178.

[57] Z. Yu, Y. Liu, X. Yu, and K. Pu, "Scalable distributed processing of K nearest neighbor queries over moving objects," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 5, pp. 1383–1396, May 2015.

[58] W. Lu, Y. Shen, S. Chen, and C. Ben, "Efficient processing of k nearest neighbor joins using MapReduce," in *Proc. Very Large Data Bases Endowment*, 2012, pp. 1016–1027.

[59] W. Kim, Y. Kim, and K. Shim, "Parallel computation of k-nearest neighbor joins using MapReduce," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 696–705.

[60] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[61] X. Cheng, X. Jin, Y. Wang, J. Guo, T. Zhang, and G. Li, "Survey on Big Data system and analytic technology," *J. Softw.*, vol. 25, no. 9, pp. 1240–1252, Jul. 2014.

[62] Z. Zhang, J. Fang, W. Chen, D. Zhang, A. Liu, and Z. Li, "QBS-tree: A spatial index with high update efficiency for real-time processing system," in *Proc. 21st Int. Conf. High Perform. Comput. Commun./IEEE 17th Int. Conf. Smart City/IEEE 5th Int. Conf. Data Sci. Syst.*, 2019, pp. 1282–1289.

[63] S. Shaik, K. Mariam, H. Kitagawa, and K. Kim, "GeoFlink: A distributed and scalable framework for the real-time processing of spatial streams," in *Proc. 29th ACM Int. Conf. Inf. Knowl. Manage.*, 2020, pp. 3149–3156.

[64] H. Wang and A. Belhassena, "Parallel trajectory search based on distributed index," *Inf. Sci.*, vol. 388–389, pp. 62–83, Jan. 2017.

[65] A. Belhassena and H. Wang, "Distributed skyline trajectory query processing," in *Proc. ACM Turing 50th Celebration Conf.-China*, 2017, pp. 1–7.

[66] R. Whitman, B. Marsh, M. Park, and E. Hoel, "Distributed spatial and spatio-temporal join on apache spark," *ACM Trans. Spatial Algorithms Syst.*, vol. 5, no. 1, pp. 1–28, Jun. 2019.

[67] Z. Shang, G. Li, and Z. Bao, "DITA: Distributed in-memory trajectory analytics," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2018, pp. 725–740.

[68] S. Hagedorn and T. Räth, "Efficient spatio-temporal event processing with STARK," in *Proc. Int. Conf. Extending Database Technol.*, 2017, pp. 570–573.

[69] J. Yu, J. Wu, and M. Sarwat, "GeoSpark: A cluster computing framework for processing large-scale spatial data," in *Proc. SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2015, pp. 1–4.
[70] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: The GeoSpark perspective and beyond," *Geoinformatica*, vol. 23, no. 1, pp. 37–78, Jan. 2019.
[71] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *Proc. IEEE 31st Int. Conf. Data Eng. Workshops*, 2015, pp. 34–41.
[72] M. Yuan *et al.*, "OceanST: A distributed analytic system for large-scale spatio-temporal mobile broadband data," in *Proc. Very Large Data Bases Endowment*, 2014, pp. 1561–1564.
[73] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1071–1085.
[74] F. Hu *et al.*, "A hierarchical indexing strategy for optimizing apache spark with HDFS to efficiently query big geospatial raster data," *Int. J. Digit. Earth*, vol. 13, no. 3, pp. 410–428, Mar. 2020.
[75] Y. Wang, Z. Gui, H. Wu, D. Peng, J. Wu, and Z. Cui, "Optimizing and accelerating spatio-temporal Ripley's K function based on apache spark for distributed spatio-temporal point pattern analysis," *Futur. Gener. Comp. Syst.*, vol. 105, pp. 96–118, Apr. 2020.
[76] J. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1981, pp. 10–18.
[77] D. Shen, G. Yu, X. Wang, T. Nie, and Y. Kou, "Survey on NoSQL for management of Big Data," *J. Softw.*, vol. 24, no. 8, pp. 1786–1803, May 2013.
[78] V. Akkineni, B. Aydin, S. Naduvil-Vadukootu, and R. Angryk, "Predictive spatio-temporal query processor on resilient distributed datasets," in *Proc. IEEE Int. Conf. Big Data Cloud Comput., Social Comput. Netw., Sustain. Comput. Commun.*, 2016, pp. 50–58.
[79] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[80] Apache HBase, 2010. [Online]. Available: https://hbase.apache.org
[81] Apache Accumulo, 2011. [Online]. Available: https://accumulo.apache.org
[82] C. Li, Z. Wu, P. Wu, and Z. Zhao, "An adaptive construction method of hierarchical index for vector data under peer-to-peer networks," *ISPRS Int. J. Geo-Inf.*, vol. 8, no. 11, pp. 1–19, Nov. 2019.
[83] Y. Ma, J. Rao, W. Hu, and X. Meng, "An efficient index for massive IOT data in cloud environment," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage.*, 2012, pp. 2129–2133.
[84] S. Nishimura, S. Das, D. Agrawal, and A. Abbadi, "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," in *Proc. IEEE 12th Int. Conf. Mobile Data Manage.*, 2011, pp. 7–16.
[85] H. Le and A. Takasu, "A scalable spatio-temporal data storage for intelligent transportation systems based on HBase," in *Proc. IEEE 18th Int. Conf. Intell. Transp. Syst.*, 2015, pp. 2733–2738.
[86] F. Liu, S. Gao, and Z. Zhou, "Research on indexing technology for AIS data stream," in *Proc. 2nd Int. Conf. Big Data Eng. Technol.*, 2020, pp. 7–12.
[87] C. Zhang, L. Zhu, J. Long, S. Lin, Z. Yang, and W. Huang, "A hybrid index model for efficient spatio-temporal search in HBase," in *Proc. Pacific-Asia Conf. Knowl. Discov. Data Mining*, 2018, pp. 108–120.
[88] N. Du, J. Zhan, M. Zhao, D. Xiao, and Y. Xie, "Spatio-temporal data index model of moving objects on fixed networks using hbase," in *Proc. IEEE Int. Conf. Comput. Intell. Commun. Technol.*, 2015, pp. 247–251.
[89] R. Li *et al.*, "JUST: JD urban spatio-temporal data engine," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2020, pp. 1558–1569.
[90] B. Yu, C. Zhang, J. Sun, and Y. Zhang, "Massive GIS spatio-temporal data storage method in cloud environment," in *Proc. 2nd Int. Conf. Comput. Sci. Artif. Intell.*, 2018, pp. 105–109.
[91] H. Xu, N. Yao, Q. Han, S. Cai, and D. Sun, "An approximate nearest neighbor query algorithm based on BZ-Tree," in *Proc. Int. Conf. Comput. Sci. Serv. Syst.*, 2012, pp. 1699–1701.
[92] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon, "Spatio-temporal indexing in non-relational distributed databases," in *Proc. IEEE Int. Conf. Big Data*, 2013, pp. 291–299.
[93] S. Park, D. Ko, and S. Song, "Parallel insertion and indexing method for large amount of spatiotemporal data using dynamic multilevel grid technique," *Appl. Sci.*, vol. 9, no. 20, Oct. 2019, Art. no. 4261.
[94] J. Hughes, A. Annex, C. Eichelberger, A. Fox, and A. Hulbert, "GeoMesa: A distributed architecture for spatio-temporal fusion," *Proc. SPIE*, vol. 9473, 2015, Art. no. 94730F.
[95] C. Böhm, G. Klump, and H. Kriegel, "XZ-Ordering: A. space-filling curve for objects with spatial extension," in *Proc. Int. Symp. Spatial Databases*, 1999, pp. 75–90.
[96] Redis, 2010. [Online]. Available: http://redis.io
[97] MongoDB, 2010. [Online]. Available: https://www.mongodb.com
[98] X. Guan, C. Bo, Z. Li, and Y. Yu, "ST-hash: An efficient spatio-temporal index for massive trajectory data in a NoSQL database," in *Proc. 25th Int. Conf. Geoinformatics*, 2017, pp. 1–7.
[99] K. Lee, M. Seo, R. Lee, M. Park, and S. Lee, "Efficient processing of spatio-temporal joins on IoT data," *IEEE Access*, vol. 8, pp. 108371–108386, 2020.
[100] S. Ke, J. Gong, S. Li, Q. Zhu, X. Liu, and Y. Zhang, "A hybrid spatio-temporal data indexing method for trajectory databases," *Sensors*, vol. 14, no. 7, pp. 12990–13005, Jul. 2014.
[101] T. Pham, D. Nguyen, and K. Doan, "S4STRD: A scalable in memory storage system for spatio-temporal real-time data," in *Proc. IEEE Int. Conf. Smart City/SocialCom/SustainCom*, 2015, pp. 896–901.
[102] C. Qian, C. Yi, C. Cheng, G. Pu, X. Wei, and H. Zhang, "GeoSOT-based spatio-temporal index of massive trajectory data," *ISPRS Int. J. Geo-Inf.*, vol. 8, no. 6, pp. 1–12, Jun. 2019.
[103] M. Zaharia *et al.*, "Apache spark: A unified engine for Big Data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.
[104] R. Cortés, X. Bonnaire, O. Marin, L. Arantes, and P. Sens, "GeoTrie: A scalable architecture for location-temporal range queries over massive geotagged data sets," in *Proc. IEEE 15th Int. Symp. Netw. Comput. Appl.*, 2016, pp. 10–17.
[105] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," in *Proc. 23rd ACM Symp. Princ. Distrib. Comput.*, 2004, pp. 368–368.
[106] A. Papadopoulos and D. Katsaros, "A-tree: Distributed indexing of multidimensional data for cloud computing environments," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 407–414.
[107] M. Malensek, S. Pallickara, and S. Pallickara, "Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals," *Futur. Gener. Comp. Syst.* vol. 29, no. 4, pp. 1049–1061, Jun. 2013.
[108] P. Memarzia, M. Patrou, M. Alam, S. Ray, V. Bhavsar, and K. Kent, "Toward efficient processing of spatio-temporal workloads in a distributed in-memory system," in *Proc. IEEE Int. Conf. Mobile Data Manage.*, 2019, pp. 118–127.
[109] M. Patrou *et al.*, "DISTIL: A distributed in-memory data processing system for location-based services," in *Proc. ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2018, pp. 496–99.
[110] A. Akdogan, C. Shahabi, and U. Demiryurek, "ToSS-it: A cloud-based throwaway spatial index structure for dynamic location data," in *Proc. IEEE 15th Int. Conf. Mobile Data Manage.*, 2014, pp. 249–258.
[111] A. Akdogan, C. Shahabi, and U. Demiryurek, "D-ToSS: A distributed throwaway spatial index structure for dynamic location data," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 9, pp. 2334–2348, Sep. 2016.
[112] Y. Zheng, L. Capra, O. Wolfson, and H. Yang, "Urban computing: Concepts, methodologies, and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 5, no. 3, pp. 1–55, Sep. 2014.

Ruijie Tian received the M.S. degree in software engineering in 2019 from Dalian Maritime University, Dalian, China, in 2019, where he is currently working toward the Ph.D. degree in computer science and technology with Information Science and Technology College.

His research interests include spatio-temporal data management, and distributed computing.

Huawei Zhai received the M.S. and Ph.D. degrees in software engineering and computer application technology from the Information Science and Technology College, Dalian Maritime University (DLMU), Dalian, China, in 2007 and 2012, respectively.

He is currently a Associate Professor with DLMU. His research interests include Big Data, intelligent transportation, and intelligent software.

Weishi Zhang received the B.S. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 1984, the M.S. degree in computer science from the Chinese Academy of Science, Beijing, China, in 1986, and the Ph.D. degree in computer science from the University of Munich, Munich, Germany, in 1996.

From 1986 to 1990, he was an Assistant Researcher with the Shenyang Institute of Computing, Chinese Academy of Science. From 1990 to 1992, he was a Visiting Scholar with Passau University, Passau, Germany. From 1992 to 1997, he was an Assistant Professor with University of Munich. In 1997, he joined the Department of Computer Science, Dalian Maritime University, Dalian, China, where he is currently a Full Professor with the School of Information Science and Technology. His research interests include computer vision, Big Data intelligent processing, software engineering, software architecture, formal specification techniques, and program semantics model.

Fei Wang received the B.S. and M.S. degrees in computer science and technology from Dalian Maritime University, Dalian, China, in 2012 and 2015, respectively, and the Ph.D. degree in control theory and engineering from the Dalian University of Technology, Dalian, China in 2019.

He is currently a Lecturer with Dalian Maritime University. His research interests include robotics, deep learning, 3-D data processing, and scene understanding.

Yao Guan received the B.S. degree in IoT engineering in 2021 from Dalian Maritime University, Dalian, China, where she is currently working toward the M.S. degree in software engineering with Information Science and Technology College.

Her research interests include spatio-temporal data management and traffic Big Data analysis.