Geoinformatica (2018) 22:785–813
https://doi.org/10.1007/s10707-018-0325-6

ST-Hadoop: a MapReduce framework for spatio-temporal data

Louai Alarabi[1] · **Mohamed F. Mokbel**[1] · **Mashaal Musleh**[1]

Received: 13 March 2018 / Accepted: 20 June 2018 / Published online: 5 July 2018

Abstract This paper presents ST-Hadoop; the first full-fledged open-source MapReduce framework with a native support for spatio-temporal data. ST-Hadoop is a comprehensive extension to Hadoop and SpatialHadoop that injects spatio-temporal data awareness inside each of their layers, mainly, language, indexing, and operations layers. In the language layer, ST-Hadoop provides built in spatio-temporal data types and operations. In the indexing layer, ST-Hadoop spatiotemporally loads and divides data across computation nodes in Hadoop Distributed File System in a way that mimics spatio-temporal index structures, which result in achieving orders of magnitude better performance than Hadoop and SpatialHadoop when dealing with spatio-temporal data and queries. In the operations layer, ST-Hadoop shipped with support for three fundamental spatio-temporal queries, namely, spatio-temporal range, top-k nearest neighbor, and join queries. Extensibility of ST-Hadoop allows others to extend features and operations easily using similar approaches described in the paper. Extensive experiments conducted on large-scale dataset of size 10 TB that contains over 1 Billion spatio-temporal records, to show that ST-Hadoop achieves orders of magnitude better performance than Hadoop and SpaitalHadoop when dealing with spatio-temporal data and operations. The key idea behind the performance gained in ST-Hadoop is its ability in indexing spatio-temporal data within Hadoop Distributed File System.

Keywords MapReduce-based systems · Spatio-temporal systems · Spatio-temporal range query · Spatio-temporal nearest neighbor query · Spatio-temporal join query

✉ Louai Alarabi
louai@cs.umn.edu

Mohamed F. Mokbel
mokbel@cs.umn.edu

Mashaal Musleh
musle005@cs.umn.edu

[1] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

1 Introduction

The importance of processing spatio-temporal data has gained much interest in the last few years, especially with the emergence and popularity of applications that create them in large-scale. For example, Taxi trajectory of New York city archive over 1.1 Billion trajectories [26], social network data (e.g., Twitter has over 500 Million new tweets every day) [1], NASA Satellite daily produces 4TB of data [23, 24], and European X-Ray Free-Electron Laser Facility produce large collection of spatio-temporal series at a rate of 40GB per second, that collectively form 50PB of data yearly [12]. Beside the huge achieved volume of the data, space and time are two fundamental characteristics that raise the demand for processing spatio-temporal data.

The current efforts to process big spatio-temporal data on MapReduce environment either use: (a) *General purpose* distributed frameworks such as Hadoop [6] or Spark [7], or (b) *Big spatial data systems* such as ESRI tools on Hadoop [33], Parallel-Secondo [20], $\mathcal{MD}$-HBase [25], Hadoop-GIS [3], GeoTrellis [17], GeoSpark [35], or SpatialHadoop [9]. The former has been acceptable for typical analysis tasks as they organize data as non-indexed heap files. However, using these systems as-is will results in sub-performance for spatio-temporal applications that need indexing [18, 22, 30]. The latter reveal their inefficiency for supporting time-varying of spatial objects because their indexes are mainly geared toward processing spatial queries, e.g., SHAHED system [10] is built on top of SpatialHadoop [9].

Even though existing big spatial systems are efficient for spatial operations, nonetheless, they suffer when they are processing spatio-temporal queries, e.g., "*find geo-tagged news in California area during the last three months*". Adopting any big spatial systems to execute common types of spatio-temporal queries, e.g., *range query*, will suffer from the following: (1) The spatial index is still ill-suited to efficiently support time-varying of spatial objects, mainly because the index are geared toward supporting spatial queries, in which result in scanning through irrelevant data to the query answer. (2) The system internal is unaware of the spatio-temporal properties of the objects, especially when they are routinely achieved in large-scale. Such aspect enforces the spatial index to be reconstructed from scratch with every batch update to accommodate new data, and thus the space division of regions in the spatial-index will be jammed, in which require more processing time for spatio-temporal queries. One possible way to recognize spatio-temporal data is to add one more dimension to the spatial index. Yet, such choice is incapable of accommodating new batch update without reconstruction the whole index from scratch.

This paper introduces ST-Hadoop; the first full-fledged open-source MapReduce framework with a native support for spatio-temporal data, available to download from [28]. ST-Hadoop is a comprehensive extension to Hadoop and SpatialHadoop that injects spatio-temporal data awareness inside each of their layers, mainly, indexing, operations, and language layers. ST-Hadoop is compatible with SpatialHadoop and Hadoop, where programs are coded as *map* and *reduce* functions. However, running a program that deals with spatio-temporal data using ST-Hadoop will have orders of magnitude better performance than Hadoop and SpatialHadoop. Figure 1a and b show how to express a spatio-temporal range query in SpatialHadoop and ST-Hadoop, respectively. The query finds all points within a certain rectangular area represented by two corner points $\langle x1, y1\rangle$, $\langle x2, y2\rangle$, and within a time interval $\langle t1,t2\rangle$. Running this query on a dataset of 10TB and a cluster of 24 nodes takes 200 seconds on SpatialHadoop as opposed to only one second on ST-Hadoop. The main reason of the sub-performance of SpatialHadoop is that it needs to scan all

```
Objects  =  LOAD 'points' AS (id:int, Location:POINT, Time:t);
Result   =  FILTER Objects BY
            Overlaps (Location, Rectangle(x1, y1, x2, y2))
             AND t < t2 AND t > t1;
```
(a) Range query in SpatialHadoop

```
Objects  =  LOAD 'points' AS (id:int, STPoint:(Location,Time));
Result   =  FILTER Objects BY
            Overlaps (STPoint, Rectangle(x1, y1, x2, y2), Interval (t1, t2) );
```
(b) Range query in ST-Hadoop

Fig. 1 Range query in SpatialHadoop vs. ST-Hadoop

the entries in its spatial index that overlap with the spatial predicate, and then check the temporal predicate of each entry individually. Meanwhile, ST-Hadoop exploits its built-in spatio-temporal index to only retrieve the data entries that overlap with *both* the spatial and temporal predicates, and hence achieves two orders of magnitude improvement over SpatialHadoop.

ST-Hadoop is a comprehensive extension of Hadoop that injects spatio-temporal awareness inside each layers of SpatialHadoop, mainly, *language*, *indexing*, *MapReduce*, and *operations* layers. In the *language* layer, ST-Hadoop extends Pigeon language [8] to supports spatio-temporal data types and operations. The *indexing* layer, ST-Hadoop spatiotemporally loads and divides data across computation nodes in the Hadoop distributed file system. In this layer ST-Hadoop scans a random sample obtained from the whole dataset, bulk loads its spatio-temporal index in-memory, and then uses the spatio-temporal boundaries of its index structure to assign data records with its overlap partitions. ST-Hadoop sacrifices storage to achieve more efficient performance in supporting spatio-temporal operations, by replicating its index into *temporal hierarchy index structure* that consists of two-layer indexing of temporal and then spatial. The *MapReduce* layer introduces two new components of *SpatioTemporalFileSplitter*, and *SpatioTemporalRecordReader*, that exploit the spatio-temporal index structures to speed up spatio-temporal operations. Finally, the *operations* layer encapsulates the spatio-temporal operations that take advantage of the ST-Hadoop *temporal hierarchy index structure* in the indexing layer, such as spatio-temporal range, nearest neighbor, and join queries.

The key idea behind the performance gain of ST-Hadoop is its ability to load the data in Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. Hence, incoming spatio-temporal queries can have minimal data access to retrieve the query answer. ST-Hadoop is shipped with support for three fundamental spatio-temporal queries, namely, spatio-temporal range, nearest neighbor, and join queries. However, ST-Hadoop is extensible to support a myriad of other spatio-temporal operations. We envision that ST-Hadoop will act as a research vehicle where developers, practitioners, and researchers worldwide, can either use it directly or enrich the system by contributing their operations and analysis techniques.

The rest of this paper is organized as follows: Section 2 highlights related work. Section 3 gives the architecture of ST-Hadoop. Details of the *language*, *spatio-temporal indexing*, and *operations* are given in Sections 4-6, followed by extensive experiments conducted in Section 8. Section 9 concludes the paper.

2 Related work

Triggered by the needs to process large-scale spatio-temporal data, there is an increasing recent interest in using Hadoop to support spatio-temporal operations. The existing work in this area can be classified and described briefly as following:

On-Top of MapReduce Framework Existing work in this category has mainly focused on addressing a specific spatio-temporal operation. The idea is to develop map and reduce functions for the required operation, which will be executed on-top of existing Hadoop cluster. Examples of these operations includes spatio-temporal range query [18, 22, 30], spatio-temporal nearest neighbor [34, 36], spatio-temporal join [4, 14, 16]. However, using Hadoop as-is results in a poor performance for spatio-temporal applications that need indexing.

Ad-hoc on Big Spatial System Several big spatial systems in this category are still ill-suited to perform spatio-temporal operations, mainly because their indexes are only geared toward processing spatial operations, and their internals are unaware of the spatio-temporal data properties [3, 9, 20, 21, 25, 29, 31, 33, 35, 37]. For example, SHAHED runs spatio-temporal operations as an ad-hoc using SpatialHadoop [9].

Spatio-temporal System Existing works in this category has mainly focused on combining the three spatio-temporal dimensions (i.e., x, y, and time) into a single-dimensional lexicographic key. For example, GeoMesa [13] and GeoWave [15, 32] both are built upon Accumulo platform [2] and implemented a space filling curve to combine the three dimensions of geometry and time. Yet, these systems do not attempt to enhance the spatial locality of data; instead they rely on time load balancing inherited by Accumulo. Hence, they will have a sup-performance for spatio-temporal operations on highly skewed data.

ST-Hadoop is designed as a generic MapReduce system to support spatio-temporal queries, and assist developers in implementing a wide selection of spatio-temporal operations. In particular, ST-Hadoop leverages the design of Hadoop and SpatialHadoop to loads and partitions data records according to their time and spatial dimension across computations nodes, which allow the parallelism of processing spatio-temporal queries when accessing its index. In this paper, we present three case study of operations that utilize the ST-Hadoop indexing, namely, spatio-temporal range, nearest neighbor, and join queries. ST-Hadoop operations achieve two or more orders of magnitude better performance, mainly because ST-Hadoop is sufficiently aware of both temporal and spatial locality of data records.

3 ST-Hadoop architecture

Figure 2 gives the high level architecture of our ST-Hadoop system; as the first full-fledged open-source MapReduce framework with a built-in support for spatio-temporal data. ST-Hadoop cluster contains one master node that breaks a map-reduce job into smaller tasks, carried out by slave nodes. Three types of users interact with ST-Hadoop: (1) *Casual users* who access ST-Hadoop through its spatio-temporal language to process their datasets. (2) *Developers*, who have a deeper understanding of the system internals and can implement new spatio-temporal operations, and (3) *Administrators*, who can tune up the system through

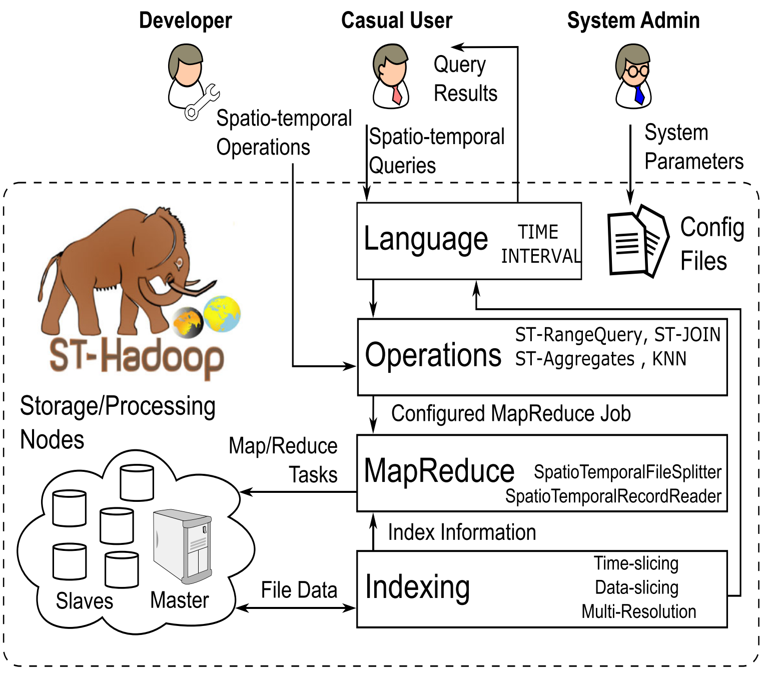


Fig. 2 ST-Hadoop system architecture

adjusting system parameters in the configuration files provided with the ST-Hadoop installation. ST-Hadoop adopts a layered design of four main layers, namely, *language*, *Indexing*, *MapReduce*, and *operations* layers, described briefly below:

Language Layer This layer extends Pigeon language [8] to supports spatio-temporal data types (i.e., STPOINT, TIME and INTERVAL) and spatio-temporal operations (e.g., OVERLAP, KNN, and JOIN). Details are given in Section 4.

Indexing Layer ST-Hadoop spatiotemporally loads and partitions data across computation nodes. In this layer ST-Hadoop scans a random sample obtained from the input dataset, bulk-loads its spatio-temporal index that consists of two-layer indexing of temporal and then spatial. Finally ST-Hadoop replicates its index into *temporal hierarchy index structure* to achieve more efficient performance for processing spatio-temporal queries. Details of the index layer are given in Section 5.

MapReduce Layer In this layer, new implementations added inside SpatialHadoop MapReduce layer to enables ST-Hadoop to exploits its spatio-temporal indexes and realizes spatio-temporal predicates. We are not going to discuss this layer any further, mainly because few changes were made to inject time awareness in this layer. The implementation of MapReduce layer was already discussed in great details [9].

Operations Layer This layer encapsulates the implementation of three common spatio-temporal operations, namely, spatio-temporal range, spatio-temporal top-k nearest neighbor, and spatio-temporal join queries. More operations can be added to this layer by ST-Hadoop *developers*. Details of the operations layer are discussed in Section 6.

4 Language layer

ST-Hadoop does not provide a completely new language. Instead, it extends Pigeon language [8] by adding spatio-temporal data types, functions, and operations. Spatio-temporal data types (STPoint, Time and Interval) are used to define the schema of input files upon their loading process. In particular, ST-Hadoop adds the following:

Data types ST-Hadoop extends `STPoint`, `TIME`, and `INTERVAL`. The `TIME` instance is used to identify the temporal dimension of the data, while the time `INTERVAL` mainly provided to equip the query predicates. The following code snippet loads NYC taxi trajectories from 'NYC' file with a column of type `STPoint`.

```
trajectory = LOAD 'NYC' as
(id:int, STPoint(loc:point, time:timestamp));
```

`NYC` and `trajectory` are the paths to the non-indexed heap file and the destination indexed file, respectively. `loc` and `time` are the columns that specify both spatial and temporal attributes.

Functions and Operations Pigeon already equipped with several basic spatial predicates. ST-Hadoop changes the `overlap` function to support spatio-temporal operations. The other predicates and their possible variation for supporting spatio-temporal data are discussed in great details in [11]. ST-Hadoop encapsulates the implementation of three commonly used spatio-temporal operations, i.e., range, nearest neighbor, and Join queries, that take the advantages of the spatio-temporal index. The following example "*retrieves all cars in State Fair area represented by its minimum boundary rectangle during the time interval of August 25th and September 6th*" from trajectory indexed file.

```
cars = FILTER trajectory
 BY overlap( STPoint,
    RECTANGLE(x1,y1,x2,y2),
    INTERVAL(08-25-2016, 09-06-2016));
```

ST-Hadoop extended the `JOIN` to take two spatio-temporal indexes as an input. The processing of the `join` invokes the corresponding spatio-temporal procedure. For example, one might need to understand the relationship between the birds death and the existence of humans around them, which can be described as "*find every pairs from birds and human trajectories that are close to each other within a distance of 1 mile during the last year*".

```
human_bird_pairs = JOIN human_trajectory, bird_trajectory
 PREDICATE = overlap( RECTANGLE(x1,y1,x2,y2),
              INTERVAL(01-01-2016, 12-31-2016),
              WITHIN_DISTANCE(1) );
```

ST-Hadoop extends `KNN` operation to finds top-k points to a given query point Q in space and time. ST-Hadoop computes the nearest neighbor proximity according to some α value that indicates whether the kNN operation leans toward spatial, temporal, or spaito-temporal closeness. The α can be any value between zero and one. A ranking function $F_\alpha(Q, p)$ computes the proximity between query point Q and any other points $p \in P$. The following code gives an example of kNN query, where a crime analyst is interested to find the relationship between crimes, which can be described as "*find the top 100 closest crimes to a given crime Q located in downtown that took place on the* 2^{nd} *during last year, with* $\alpha = 0.3$".

```
k_crimes = KNN crimes_data
 PREDICATE = WITH_K=100
             WITH_alpha=0.3
             USING F(Q, crime);
```

5 Indexing layer

Input files in Hadoop Distributed File System (HDFS) are organized as a heap structure, where the input is partitioned into chunks, each of size 64MB. Given a file, the first 64MB is loaded to one partition, then the second 64MB is loaded in a second partition, and so on. While that was acceptable for typical Hadoop applications (e.g., analysis tasks), it will not support spatio-temporal applications where there is always a need to filter input data with spatial and temporal predicates. Meanwhile, spatially indexed HDFSs, as in SpatialHadoop [9] and ScalaGiST [21], are geared towards queries with spatial predicates only. This means that a temporal query to these systems will need to scan the whole dataset. Also, a spatio-temporal query with a small temporal predicate may end up scanning large amounts of data. For example, consider an input file that includes all social media contents in the whole world for the last five years or so. A query that asks about contents in the USA in a certain hour may end up in scanning all the five years contents of the USA to find out the answer.

ST-Hadoop HDFS organizes input files as spatio-temporal partitions that satisfy one main goal of supporting spatio-temporal queries. ST-Hadoop imposes temporal slicing, where input files are spatiotemporally loaded into intervals of a specific time granularity, e.g., days, weeks, or months. Each granularity is represented as a level in ST-Hadoop index. Data records in each level are spatiotemporally partitioned, such that the boundary of a partition is defined by a spatial region and time interval.

Figures 3a and b show the HDFS organization in SpatialHadoop and ST-Hadoop frameworks, respectively. Rectangular shapes represent boundaries of the HDFS partitions within

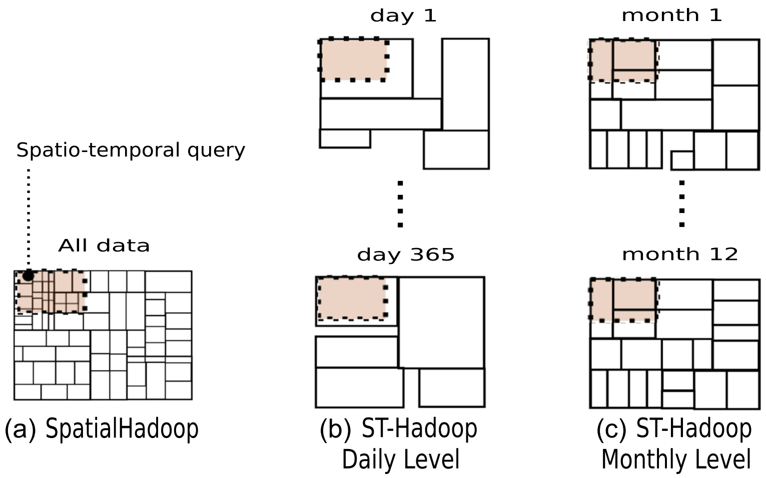


Fig. 3 HDFSs in ST-Hadoop VS SpatialHadoop

their framework, where each partition maintains a 64MB of nearby objects. The dotted square is an example of a spatio-temporal range query. For simplicity, let's consider a one year of spatio-temporal records loaded to both frameworks. As shown in Fig. 3a, SpatialHadoop is unaware of the temporal locality of the data, and thus, all records will be loaded once and partitioned according to their existence in the space. Meanwhile in Fig. 3b, ST-Hadoop loads and partitions data records for each day of the year individually, such that each partition maintains a 64MB of objects that are close to each other in both space and time. Note that HDFS partitions in both frameworks vary in their boundaries, mainly because spatial and temporal locality of objects are not the same over time. Let's assume the spatio-temporal query in the dotted square "*find objects in a certain spatial region during a specific month*" in Fig. 3a, and b. SpatialHadoop needs to access all partitions overlapped with query region, and hence SpatialHadoop is required to scan one year of records to get the final answer. In the meantime, ST-Hadoop reports the query answer by accessing few partitions from its daily level without the need to scan a huge number of records.

5.1 Concept of hierarchy

ST-Hadoop imposes a replication of data to support spatio-temporal queries with different granularities. The data replication is reasonable as the storage in ST-Hadoop cluster is inexpensive, and thus, sacrificing storage to gain more efficient performance is not a drawback. Updates are not a problem with replication, mainly because ST-Hadoop extends MapReduce framework that is essentially designed for batch processing, thereby ST-Hadoop utilizes incremental batch accommodation for new updates.

The key idea behind the performance gain of ST-Hadoop is its ability to load the data in Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. To support all spatio-temporal operations including more sophisticated queries over time, ST-Hadoop replicates spatio-temporal data into a *Temporal Hierarchy Index*. Figure 3b and c depict two levels of days and months in ST-Hadoop index structure. The same data is replicated on both levels, but with different spatio-temporal granularities. For example, a spatio-temporal query asks for objects in one month could be reported from any level in ST-Hadoop index. However, rather than hitting 30 days' partitions from the daily-level, it will be much faster to access less number of partitions by obtaining the answer from one month in the monthly-level.

A system parameter can be tuned by ST-Hadoop administrator to choose the number of levels in the *Temporal Hierarchy index*. By default, ST-Hadoop set its index structure to four levels of days, weeks, months and years granularities. However, ST-Hadoop users can easily change the granularity of any level. For example, the following code loads taxi trajectory dataset from "NYC" file using one-hour granularity, Where the `Level` and `Granularity` are two parameters that indicate which level and the desired granularity, respectively.

```
trajectory = LOAD 'NYC' as
           (id:int, STPoint(loc:point, time:timestamp))
           Level:1 Granularity:1-hour;
```

5.2 Index construction

Figure 4 illustrates the indexing construction in ST-Hadoop, which involves two scanning processes. The first process starts by scanning input files to get a random sample, and this is essential because the size of input files is beyond memory capacity, and thus, ST-Hadoop

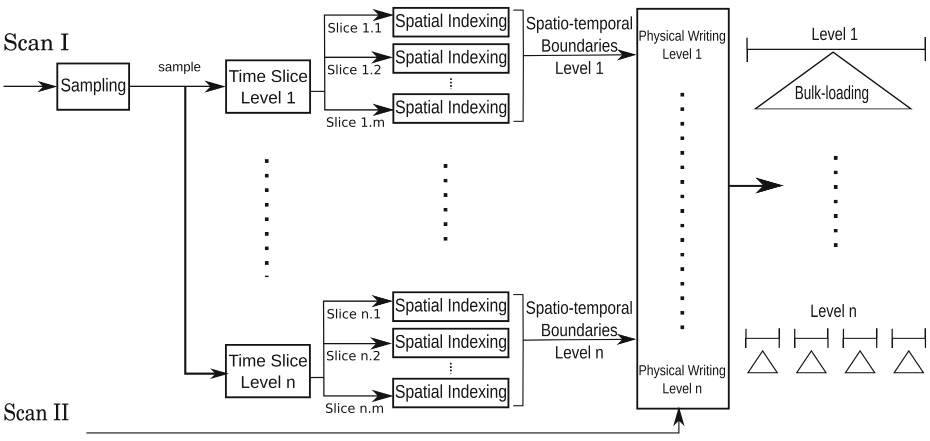


Fig. 4 Indexing in ST-Hadoop

obtains a set of records to a sample that can fit in memory. Next, ST-Hadoop processes the sample n times, where n is the number of levels in ST-Hadoop index structure. The temporal slicing in each level splits the sample into m number of slice (e.g., $slice_{1.m}$). ST-Hadoop finds the spatio-temporal boundaries by applying a spatial indexing on each temporal slice individually. As a result, outputs from temporal slicing and spatial indexing collectively represent the spatio-temporal boundaries of ST-Hadoop index structure. These boundaries will be stored as meta-data on the master node to guide the next process. The second scanning process physically assigns data records in the input files with its overlapping spatio-temporal boundaries. Note that each record in the dataset will be assigned n times, according to the number of levels.

ST-Hadoop index consists of two-layer indexing of a temporal and spatial. The conceptual visualization of the index is shown in the right of Fig. 4, where lines signify how the temporal index divided the sample into a set of disjoint time intervals, and triangles symbolize the spatial indexing. This two-layer indexing is replicated in all levels, where in each level the sample is partitioned using different granularity. ST-Hadoop trade-off storage to achieve more efficient performance through its index replication. In general, the index creation of a single level in the *Temporal Hierarchy* goes through four consecutive phases, namely sampling, temporal slicing, spatial indexing, and physical writing.

5.3 Phase I: Sampling

The objective of this phase is to approximate the spatial distribution of objects and how that distribution evolves over time, to ensure the quality of indexing; and thus, enhance the query performance. This phase is necessary, mainly because the input files are too large to fit in memory. ST-Hadoop employs a map-reduce job to efficiently read a sample through scanning all data records. We fit the sample into an in-memory simple data structure of a length (L), that is an equal to the number of HDFS blocks, which can be directly calculated from the equation $L = (Z/B)$, where Z is the total size of input files, and B is the HDFS block capacity (e.g., 64MB). The size of the random sample is set to a default ratio of 1% of input files, with a maximum size that fits in the memory of the master node. This simple data structure represented as a collection of elements; each element consist of a time instance and a space sampling that describe the time interval and the spatial distribution of spatio-temporal objects, respectively. Once the sample is scanned, we sort the sample elements in chronological order to their time instance, and thus the sample approximates the spatio-temporal distribution of input files.

5.4 Phase II: Temporal slicing

In this phase ST-Hadoop determines the temporal boundaries by slicing the in-memory sample into multiple time intervals, to efficiently support a fast random access to a sequence of objects bounded by the same time interval. ST-Hadoop employs two temporal slicing techniques, where each manipulates the sample according to specific slicing characteristics: (1) *Time-partition*, slices the sample into multiple splits that are uniformly on their time intervals, and (2) *Data-partition* where the sample is sliced to the degree that all sub-splits are uniformly in their data size. The output of this phase finds the temporal boundary of each split, that collectively cover the whole time domain.

The rational reason behind ST-Hadoop two temporal slicing techniques is that for some spatio-temporal archive the data spans a long time-interval such as decades, but their size is moderated compared to other archives that are daily collect terabytes or petabytes of spatio-temporal records. ST-Hadoop proposed the two techniques to slice the time dimension of input files based on either time-partition or data-partition, to improve the indexing quality, and thus gain efficient query performance. The time-partition slicing technique serves best in a situation where data records are uniformly distributed in time. Meanwhile, data-partition slicing best suited with data that are sparse in their time dimension.

- *Data-partition Slicing.* The goal of this approach is to slice the sample to the degree that all sub-splits are equally in their size. Figure 5 depicts the key concept of this slicing technique, such that a $\texttt{slice}_1$ and $\texttt{slice}_n$ are equally in size, while they differ in their interval coverage. In particular, the temporal boundary of $\texttt{slice}_1$ spans more time interval than $\texttt{slice}_n$. For example, consider 128MB as the size of HDFS block and input files of 1 TB. Typically, the data will be loaded into 8 thousand blocks. To load these blocks into ten equally balanced slices, ST-Hadoop first reads a sample, then sort the sample, and apply `Data-partition` technique that slices data into multiple splits. Each split contains around 800 blocks, which hold roughly a 100 GB of spatio-temporal records. There might be a small variance in size between slices, which is expectable. Similarly, another level in ST-Hadoop temporal hierarchy index could loads the 1 TB into 20 equally balanced slices, where each slice contains around 400 HDFS blocks. ST-Hadoop users are allowed to specify the granularity of data slicing by tuning α parameter. By default four ratios of α is set to 1%, 10%, 25%, and 50% that create the four levels in ST-Hadoop index structure.
- *Time-partition Slicing.* The ultimate goal of this approach is to slices the input files into multiple HDFS chunks with a specified interval. Figure 6 shows the general idea, where ST-Hadoop splits the input files into an interval of one-month granularity. While the time interval of the slices is fixed, the size of data within slices might vary. For example, as shown in Fig. 6 Jan slice has more HDFS blocks than April.

  ST-Hadoop users are allowed to specify the granularity of this slicing technique, which specified the time boundaries of all splits. By default, ST-Hadoop finer granularity level is set to one-day. Since the granularity of the slicing is known, then a straightforward solution is to find the minimum and maximum time instance of the

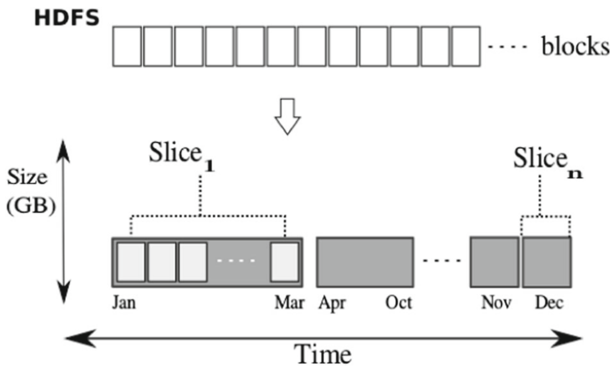


Fig. 5 Data-Slice

sample, and then based on the intervals between the both times ST-Hadoop hashes elements in the sample to the desired granularity. The number of slices generated by the time-partition technique will highly depend on the intervals between the minimum and the maximum times obtained from the sample. By default, ST-Hadoop set its index structure to four levels of days, weeks, months and years granularities.

5.5 Phase III: Spatial indexing

This phase ST-Hadoop determines the spatial boundaries of the data records within each temporal slice. ST-Hadoop spatially index each temporal slice independently; such decision handles a case where there is a significant disparity in the spatial distribution between slices, and also to preserve the spatial locality of data records. Using the same sample from the previous phase, ST-Hadoop takes the advantages of applying different types of spatial bulk loading techniques in HDFS that are already implemented in SpatialHadoop such as Grid, R-tree, Quad-tree, and Kd-tree. The output of this phase is the spatio-temporal boundaries of each temporal slice. These boundaries stored as a meta-data in a file on the master node of ST-Hadoop cluster. Each entry in the meta-data represents a partition, such as $< id, MBR, interval, level >$. Where id is a unique identifier number of a partition on

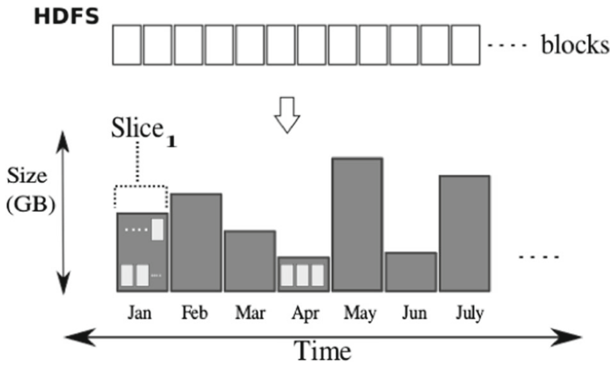


Fig. 6 Time-Slice

the HDFS, MBR is the spatial minimum boundary rectangle, $interval$ is the time boundary, and the level is the number that indicates which level in ST-Hadoop temporal hierarchy index.

5.6 Phase IV: Physical writing

Given the spatio-temporal boundaries that represent all HDFS partitions, we initiate a map-reduce job that scans through the input files and physically partitions HDFS block, by assign data records to overlapping partitions according to the spatio-temporal boundaries in the meta-data stored on the master node of ST-Hadoop cluster. For each record r assigned to a partition p, the map function writes an intermediate pair $\langle p, r\rangle$ Such pairs are then grouped by p and sent to the reduce function to write the physical partition to the HDFS. Note that for a record r will be assigned n times, depends on the number of levels in ST-Hadoop index.

6 Operations layer

The combination of the spatiotemporally load balancing with the temporal hierarchy index structure gives the core of ST-Hadoop, that enables the possibility of efficient and practical realization of spatio-temporal operations, and hence provides orders of magnitude better performance over Hadoop and SpatialHadoop. In this section, we discuss several fundamental spatio-temporal operations, namely, range (Section 6.1), *k*NN (Section 6.2), and join (Sections 6.3) as case studies of how to exploit the spatio-temporal indexing in ST-Hadoop. Other operations can also be realized following similar approaches.

6.1 Spatio-temporal range query

A range query is specified by two predicates of a spatial area and a temporal interval, A and T, respectively. The query finds a set of records R that overlap with both a region A and a time interval T, such as *"finding geotagged news in California area during the last three months"*. ST-Hadoop employs its spatio-temporal index described in Section 5 to provide an efficient algorithm that runs in three steps, *temporal filtering*, *spatial search*, and *spatio-temporal refinement*, described below.

In the **temporal filtering** step, the hierarchy index is examined to select a subset of partitions that cover the temporal interval T. The main challenge in this step is that the partitions in each granularity cover the whole time and space, which means the query can be answered from any level individually or we can mix and match partitions from different level to cover the query interval T. Depending on which granularities are used to cover T, there is a tradeoff between the number of matched partitions and the amount of processing needed to process each partition. To decide whether a partition P is selected or not, ST-Hadoop computes the coverage ratio along with the number of partitions needed to be processed and then selects the granularity based on the minimum number of partitions.

In the **spatial search** step, Once the temporal partitions are selected, the *spatial search* step applies the spatial range query against each matched partition to select records that spatially match the query range A. Keep in mind that each partition is spatiotemporally indexed which makes queries run very efficiently. Since these partitions are indexed independently, they can all be processed simultaneously across computation nodes in ST-Hadoop, and thus maximizes the computing utilization of the machines.

Finally in the **spatio-temporal refinement** step, compares individual records returned by the *spatial search* step against the query interval T, to select the exact matching records. This step is required as some of the selected temporal partitions might partially overlap the query interval T and they need to be refined to remove records that are outside T. Similarly, there is a chance that selected partitions might partially overlap with the query area A, and thus records outside the A need to be excluded from the final answer.

6.2 Spatio-temporal *k*NN query

The spatio-temporal nearest neighbor query takes a spatio-temporal point Q, a spatio-temporal predicates θ, a spatio-temporal ranking function F_α, and an integer k as an input, and returns the k spatiotemporally closest points to Q such that: (1) The k points are within the temporal distance θ_{time}. (2) The k points are not far from the spatial distance θ_{space}. (3) The top k points are ranked according to the spatio-temporal ranking function F_α that combines the spatial proximity and the temporal closeness of $p \in P$ to the query point Q. For example, a crime analyst might be interested to find the relationship between crimes, which can be described as "*find the top 10 closest crimes to a given crime Q in downtown that took place on the* 2^{nd} *during last year*". With the spatio-temporal information of the query point Q, ST-Hadoop adds a spatio-temporal ranking function F_α to the *k*NN query. The ranking function allows ST-Hadoop to compromise between spatial proximity and temporal closeness of its top-k points to the the query point.

Definition Spatio-temporal Ranking Function The ranking function F_α indicates whether a user query leans toward spatial proximity or temporal concurrency. If $\alpha = 1$, then the user cares about spatial closeness, i.e., the top-k results will be spatially closest to the query point. If $\alpha = 0$, then the user cares about temporal recency, i.e., the top-k results will be temporally recent to query point. Meanwhile, if α value is between zero and one, then the user cares about spatio-temporal proximity. The spatio-temporal proximity can be computed with the following mathematical equation.

$$\begin{aligned} F_\alpha(Q, p) = {} & \alpha \times SpatialDist(Q.loction, p.location) \\ & + (1-\alpha) \times TemporalDist(Q.timestamp, p.timestamp) \end{aligned}$$

The spatio-temporal ranking function F_α dependents on both *SpatialDist* and the *TemporalDist* functions, which they are normalized and monotonic. Each has a value range from zero to one. The *SpatialDist* is the Euclidean distance between two points' locations divided by the maximum spatial distance θ_{space}, where θ_{space} is the distance from a query point Q to the k^{th} furthest location. Meanwhile, the *TemporalDist* is that ratio of delta times of Q and p to the total temporal interval θ_{time}. The temporal interval θ_{time} is the time distance from the query point Q to the k^{th} furthest point in time.

Figure 7 gives a landscape of all possible ways to process the *k*NN operation in ST-Hadoop. Without loss of generality, let's suppose that the ST-Hadoop indexes input files into intervals of days, and a user is interested in discovering the top k points to a given query point Q during the last year. As shown in the top of the Figure, One extreme when α is equal to one, which indicates that the user cares about spatial proximity in their k result. Hence, all partitions overlap with query point Q needs to be processed from the last year. On the opposite side if α is equal to zero, then the user cares about temporal closeness in their k result. This means that at most we are going to process partitions at the same time interval. Between those two extremes reside the challenge, such that to what extent we need to process partitions from other time intervals to find the top-k points. ST-Hadoop applies

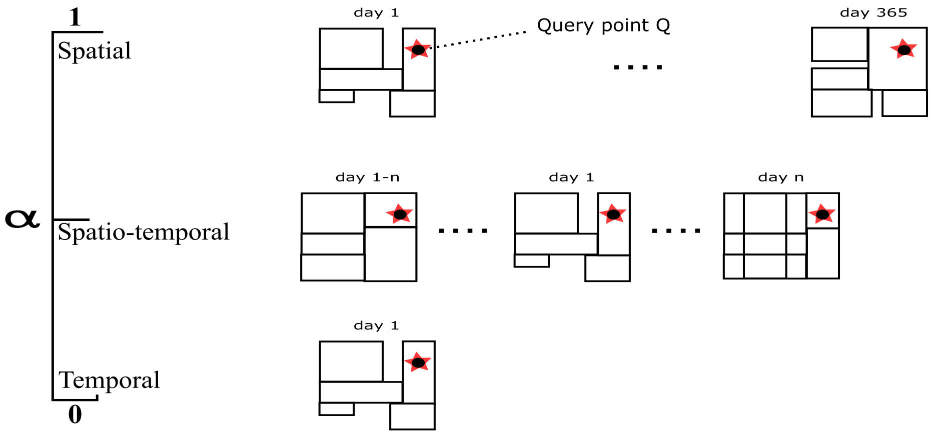


Fig. 7 Landscape of spatio-temporal *k*NN operation

a simple and efficient technique that capable of pruning the search space to process only n number of partitions, which guarantee that those partitions will have the final k answers.

In Hadoop, a *k*NN query scans entire points in input files, calculates their spatio-temporal similarity distance to the query point Q, and provides the top-k to Q [34, 36]. Meanwhile, in spatially indexed HDFS's [9, 21, 33, 38], only spatial *k*NN is supported. This means that the *k*NN operation on a spatially indexed HDFS also needs to scan all points in input files to search for the temporal closeness. ST-Hadoop considers both space and time; and thus, in its spatio-temporal *k*NN query exploits simple pruning techniques to achieve orders of magnitude better performance. ST-Hadoop *k*NN algorithm runs in three phases, *k*NN initial answer, correctness check, and *k*NN refinement.

In the ***k*NN initial answer phase**, we come up with an initial answer of the k closest points to Q within a single partition in the HDFS. ST-Hadoop first locates the partition that includes Q, by feeding the *SpatioTmeporalFileSplitter* with a filter function that selects only the overlapping partition from the temporal interval. ST-Hadoop exploits a *SpatioTmeporalRecordReader* to reads the selected partition, then executes a traditional *k*NN algorithm to produce the initial k answers. The function F_α computes the spatio-temporal distance between any points and the query point Q.

In the **correctness check phase**, we check if the initial k answer can be considered final. The main idea of this phase is to draw a test Cylinder centered at Q with radius r equal to the spatial distance from Q to its k^{th} furthest neighbor in space. The height l of the cylinder is equal to the temporal distance from Q to its k^{th} furthest neighbor in time. The radius and the height of the cylinder change only if there is potential point dominate the score of the ranking functions of the furthest k^{th} point in the initial results in any dimension, i.e., space or time. If the cylinder does not overlap with any partitions spatially or temporally, then we terminate the process, and the initial answer is considered final. Otherwise, we proceed to the next phase.

Three cases encounter when we draw the test cylinder to check for correctness in this phase as follows.

- **Case 1** ($\alpha = 1$)**:** If a user specifies α with 1, then the user cares more about spatial proximity than temporal. This means that we need to check the correctness of all time intervals. As shown in the top of Fig. 7, the query point Q overlap with all year partitions. If input files only indexed in one level, then the cylinder height is equal to the whole θ_{time}. On the other hand, if the input files indexed into a temporal hierarchy, then rather than accessing a huge number of partitions, ST-Hadoop feeds the temporal query predicate θ_{time} to its query optimizer. The query optimizer will generate an execution plan that selects the overlap partition with Q from a lower granularity level, i.e., yearly level. Next, we execute a traditional kNN algorithm to produce new initial k answers again. The new height of the cylinder is going to be equal to zero. Next, we draw a cylinder centered at Q with a radius equal to the furthest k^{th} neighbor. If the cylinder does not overlap with any partition other than Q, then we terminate, and the new initial answer considered final. Otherwise, we processed to the next phase (Fig. 8).

- **Case 2** ($\alpha = 0$)**:** If a user specifies α with zero, then the user cares more about temporal proximity. This means that we need to check the correctness from the same time interval. First, we draw a cylinder centered at Q with a radius equal to the spatial distance from Q to its k^{th} furthest neighbor, obtained from the initial answer. The height of the cylinder is equal to zero. If the cylinder does not overlap with any partition other than Q, then we terminate the process, and the initial answer considered final. Otherwise, we processed to the next phase.

Figure 9 gives an example of a kNN query for point Q with a $k = 3$ and α is equal to zero. The shaded partitions are the one considered in the processing. The dotted test cylinder has a height equal to zero, composed from the initial answer p1, p5, p16. The cylinder does not overlap with any other partitions than Q; thus, the initial answer is considered final.

- **Case 3** ($0 \leq \alpha \leq 1$)**:** If a user specifies any α value between zero and one, then this means that the user cares about the spatio-temporal proximity. The main idea is to gradually draw the cylinder and make sure that the k^{th} furthest neighbor point is not dominated by any other points in both dimensions, i.e., space and time. A point dominates the k^{th} point if it is as good or better in ranking score, and better at least in one dimension of either spatial or temporal.

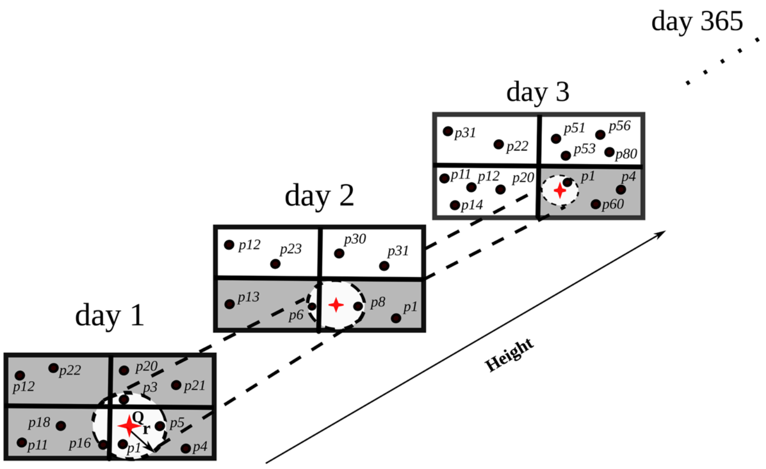


Fig. 8 Correctness Check when $0 \leq \alpha \leq 1$

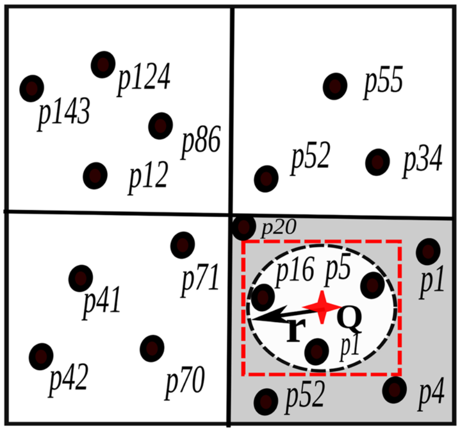


Fig. 9 Correctness check Final Answer

Figure 8 illustrates the idea of test cylinder. The query point Q initially overlap with a single time interval, e.g., day 1. We check if some points either in the next or previous interval can dominate the score of the k^{th} furthest neighbor from the same initial interval, e.g., day 1. If a dominance point exists, then we modify the cylinder height and radius accordingly in the next interval. Notice that the radius of the cylinder in next time interval is getting smaller, this is because we gradually draw the test cylinder. We continue this process until we reach a time interval that has no dominance point that can dominate the k^{th} furthest point.

The cautiously drawing of the test cylinder algorithm has two consecutive steps as follows.

Step I: In this step, we check if there is a point from a different partition(s) exist within the same temporal interval, such that it can dominate the ranking score of the initial furthest k^{th} neighbor. In other words, in this step, we determine the radius of the cylinder within the same temporal interval. First, Starting from the partition that overlap with Q from the same time interval, we draw the circle of the cylinder centered at Q with a radius equal to the spatial distance of the k^{th} furthest neighbor. If partitions overlap with the circle's MBR, then we check if the nearest point from the overlapped partition(s) can dominate the score of the initial furthest k^{th}. If a dominance exists, then we consider processing this partition, update our furthest k^{th} in the initial answer with the dominating point, and subsequently, we proceed to the next step.

Step II: In this step, we modify the height of the test cylinder by checking if there is a point from the next or the previous temporal interval can dominate the furthest k^{th} neighbor. For the sake of simplicity, let's consider the next temporal interval in our discussion. However, the presented technique is operated to examine interval in both directions. First, we find the partition that overlaps with Q from the next time interval. Then, we check if the temporal distance along with the minimum spatial distance between Q and the new partition can dominate the furthest k^{th}. If the score beats the k^{th}, then a dominance might exist in that partition. Thus, we consider processing this partition by modifying the height of the cylinder. Recursively we repeat the processing of the two steps with every new interval appended to the cylinder height until no further dominance exist. Finally, if no dominance point exists, then we can proceed to the next phase.

Figure 8 gives an example of a *k*NN query that finds the top-4 points for point Q with α value equal to 0.2 over the last year. ST-Hadoop starts from the Q partition on day 1. First, we find the top-4 neighbor from day 1, and then we insert the score of the furthest 4^{th} neighbor from day 1 to a priority queue, e.g., p_{16}. Iterate over the other overlap partitions from next temporal interval, e.g., days 2. In each iteration ST-Hadoop checks if the ranking score of the minimum distance point of the new partition can beat the ranking score of the p_{16}. If it beats, then this new partition is considered, and we modify the cylinder height and radius respectively. As depicted in Fig. 8, p_6 in day 2 dominates p_{16}. We repeat this process until no dominance point can be found in the next temporal interval. As shown in the example after the third day, we do not need to modify the height of the cylinder, since there is no dominance point exist any further. Henceforward, other partitions will be ignored and no further computation required, and we can proceed to the next phase.

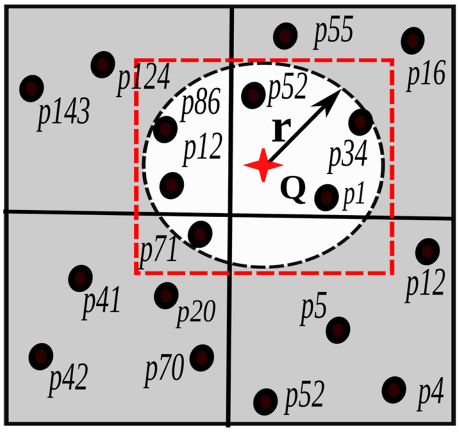
In the ***k*NN refinement phase**, We check if there are points in the overlap partitions might contribute to the final answer. If α is equal to zero or one, then we run a spatial range query to get all points inside the MBR of the test circle, as the cylinder height is equal to zero. Meanwhile, if $0 \leq \alpha \leq 1$, then we run we run a spatio-temporal range query to get all points inside the MBR of the test cylinder. The cylinder radius and height are obtained from the previous phase. Finally, we scan over the range query result and process it with the traditional *k*NN algorithm to find the final answer.

Figures 10 and 8 gives two examples of refinement phase. The shaded partitions are the ones that are considered in the range query. In Fig. 10, the circle of the cylinder intersects with the three partitions from the same temporal interval. In that case, the height of the cylinder is equal to zero. In Fig. 8 we illustrate the test cylinder with a height equal to 3 temporal intervals. Similarly, the shaded partitions are the only one to be considered in the range query. For each time interval, a circle has a different radius, which collectively forms the test cylinder. Once we get the results, we apply the traditional *k*NN algorithm to find the final top-4 answers, i.e., $\{p5, p3, p6, p1\}$ in the refinement phase.

6.3 Spatio-temporal join

Given two indexed dataset R and S of spatio-temporal records, and a spatio-temporal predicate θ. The join operation retrieves all pairs of records $\langle r, s \rangle$ that are similar to each other based on θ. For example, one might need to understand the relationship between the birds

Fig. 10 Refinement Final Answer



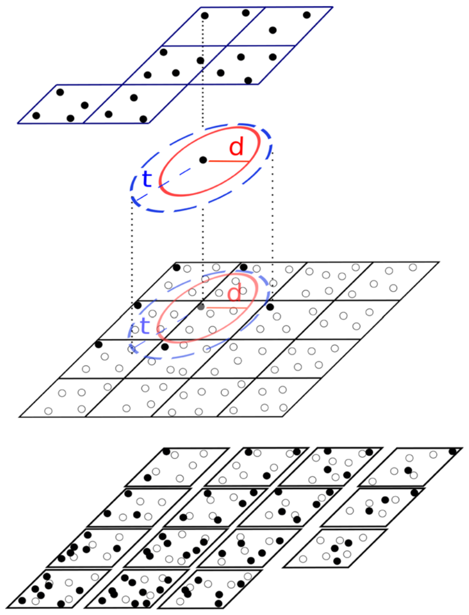


Fig. 11 Spatio-temporal Join

death and the existence of humans around them, which can be described as *"find every pairs from bird and human trajectories that are close to each other within a distance of 1 mile during the last week"*. The join algorithm runs in two steps as shown in Fig. 11, *hash* and *join*.

In the **hashing** step, the map function scans the two input files and hashes each record to candidate buckets. The buckets are defined by partitioning the spatio-temporal space using the two-layer indexing of temporal and spatial, respectively. The granularity of the partitioning controls the tradeoff between partitioning overhead and load balance, where a more granular-partitioning increases the replication overhead, but improves the load balance due to the huge number of partitions, while a less granular-partitioning minimizes the replication overhead, but can result in a huge imbalance especially with highly skewed data. The hash function assigns each point in the left dataset, $r \in R$, to all buckets within an Euclidean distance d and temporal distance t, and assigns each point in the right dataset, $s \in S$, to the one bucket which encloses the point s. This ensures that a pair of matching records $\langle r, s\rangle$ are assigned to at least one common bucket. Replication of only one dataset (R) along with the use of single assignment, ensure that the answer contains no replicas.

In the **joining** step, each bucket is assigned to one reducer that performs a traditional in-memory spatio-temporal join of the two assigned sets of records from R and S. We use the plane-sweep algorithm which can be generalized to multidimensional space. The set S is not replicated, as each pair is generated by exactly one reducer, and thus no *duplicate avoidance* step is necessary.

7 Query optimization

Figure 12 illustrates the conceptual visualization of ST-Hadoop index, where lines signify how the temporal index divided into a set of disjoint time intervals, e.g., months, weeks,

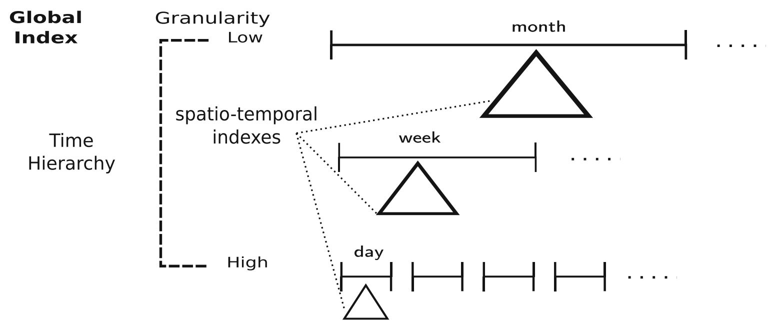


Fig. 12 ST-Hadoop spatio-temporal meta-data index

or days. Triangles symbolize the spatial indexing, e.g., R-tree. ST-Hadoop stores its index information as a meta-data on the master node. The meta-data of the ST-Hadoop index provides a rich statistics about the spatial and temporal locality of partitions in the HDFS. Each record of the meta-data represents a partition, which contains information about the minimum boundary rectangle, temporal interval, temporal granularity (i.e., level), the number of data records within the HDFS block, and a unique identifier that acts as an entry pointer to access the partition block. Data records are replicated and spatiotemporally partitioned in each level.

Partitions in each level of ST-Hadoop index cover the whole time and space, which means a query can be answered from any level individually or we can mix and match partitions from different levels to cover the query predicates. To decide which partitions should be selected and from which levels, highly depends on the selectivity of the query predicates. Depending on which granularity is used to get the result, there is a tradeoff between the number of partitions contains the query results and the amount of processing needed to process each partition. The HDFS block size is tuned in ST-Hadoop configuration files, which means partitions block size are the same across all computation nodes. Therefore, for any given query the bottleneck that hits the query performance is the number of partitions that contain the query answer.

The primary goal of ST-Hadoop query optimizer is to minimize the number of partitions that contain the final answer for each of its operation. We implemented in memory greedy algorithm that recursively iterates over ST-Hadoop meta-data to find the optimal execution plan. The algorithm runs in a top-down approach that starts from the lowermost granularity (e.g., monthly level) to the highest one (e.g., daily level). In each iteration, we examine the precise number of partitions N that contains the final answer for the given spatio-temporal query predicates. The algorithm reports the global optimal execution plan, if the next granularity has a higher number of partitions, or it reaches the highest level. For example, consider a query that asks about 22 days of data records in a particular area. First, ST-Hadoop calculates the number overlap partitions from the monthly level, and then compares it with the next level from its temporal hierarchy index (e.g., week). ST-Hadoop query optimizer recursively computes and compares the number of partitions until the next explored level has more partitions from the current one. If the highest granularity is reached and has a fewer number of partitions, then all partitions overlap with query predicates will be selected.

ST-Hadoop previous query optimizer was based on heuristic [5]. An algorithm applied to computes the *coverage ratio* r, that defined as the ratio of the time interval of a partition that overlaps with spatio-temporal query predicates. A partition was selected only if its *coverage ratio* is above a specific threshold $\mathcal{M}$. The algorithm run in a top-down approach that starts with the top level and selects partitions that cover the temporal query interval T, If the query interval T is not covered at that granularity, then the algorithm continues to the next level. If the bottom level is reached, then all partitions overlap with T will be selected. Meanwhile, in our new ST-Hadoop algorithm we employ a greedy algorithm that finds the minimum number of partitions need to be processed.

8 Experiments

This section provides an extensive experimental performance study of ST-Hadoop compared to SpatialHadoop and Hadoop. We decided to compare with this two frameworks and not other spatio-temporal DBMSs for two reasons. First, as our contributions are all about spatio-temporal data support in Hadoop. Second, the different architectures of spatio-temporal DBMSs have great influence on their respective performance, which is out of the scope of this paper. Interested readers can refer to a previous study [27] which has been established to compare different large-scale data analysis architectures. In other words, ST-Hadoop is targeted for Hadoop users who would like to process large-scale spatio-temporal data but are not satisfied with its performance. The experiments are designed to show the effect of ST-Hadoop indexing and the overhead imposed by its new features compared to SpatialHadoop. However, ST-Hadoop achieves two orders of magnitude improvement over SpatialHadoop and Hadoop.

Experimental Settings All experiments are conducted on a dedicated internal cluster of 24 nodes. Each has 64GB memory, 2TB storage, and Intel(R) Xeon(R) CPU 3GHz of 8 core processor. We use Hadoop 2.7.2 running on Java 1.7 and Ubuntu 14.04.5 LTS. Figure 13b summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless mentioned.

Datasets To test the performance of ST-Hadoop we use the Twitter archived dataset [1]. The dataset collected using the public Twitter API for more than three years, which contains over 1 Billion spatio-temporal records with a total size of 10TB. To scale out time in our experiments we divided the dataset into different time intervals and sizes, respectively as shown in Fig. 13a. The default size used is 1TB which is big enough for our extensive experiments unless mentioned.

Twitter Data	Size	Num-Records	Time window
Large	10TB	> 1 Billion	> 3 years
Average-Large	6.7TB	692 Million	1 years
Medium-Large	3TB	152 Million	9 months
Moderate-Large	(1TB)	115 Million	3 months

(a) Datasets

Parameter	Values (default)
HDFS block capacity (B)	32, 64, (128), 256 MB
Cluster size (N)	5, 10, 15, 20, (23)
Selection ratio (ρ)	(0.01), 0.02, 0.05, 0.1, 0.2, 0.5, 1.0
Data-partition slicing ratio(α)	0.01, 0.02, 0.025, 0.05, (0.1), 1
Time-partition slicing granularity(σ)	(days), weeks, months, years
Spatio-temporal proximity (α)	0,0.2, (0.5), 0.6, 0.8, 1.0

(b) Parameters

Fig. 13 Experimental settings and Dataset

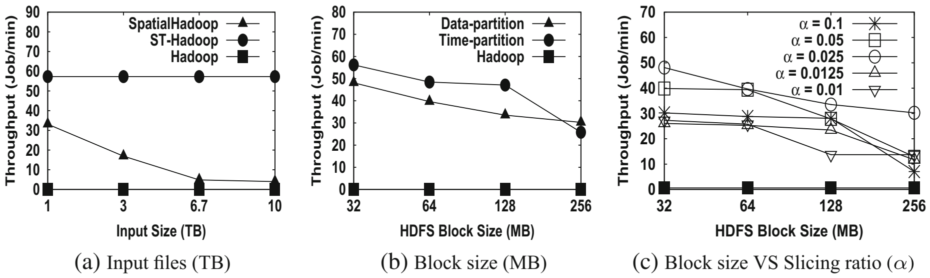


Fig. 14 Spatio-temporal Range Query

In our experiments, we compare the performance of a ST-Hadoop spatio-temporal range and join query proposed in Section 6 to their spatial-temporal implementations on-top of SpatialHadoop and Hadoop. For range query, we use system throughput as the performance metric, which indicates the number of MapReduce jobs finished per minute. To calculate the throughput, a batch of 20 queries is submitted to the system, and the throughput is calculated by dividing 20 by the total time of all queries. The 20 queries are randomly selected with a spatial area ratio of 0.001% and a temporal window of 24 hours unless stated. This experimental design ensures that all machines get busy and the cluster stays fully utilized. For spatio-temporal join, we use the processing time of one query as the performance metric as one query is usually enough to keep all machines busy. The experimental results for range, *k*NN, and join queries are reported in Sections 8.1, 8.2 and 8.3, respectively. Meanwhile, Section 8.4 evaluates our query optimizer, and Section 8.5 analyzes ST-Hadoop indexing.

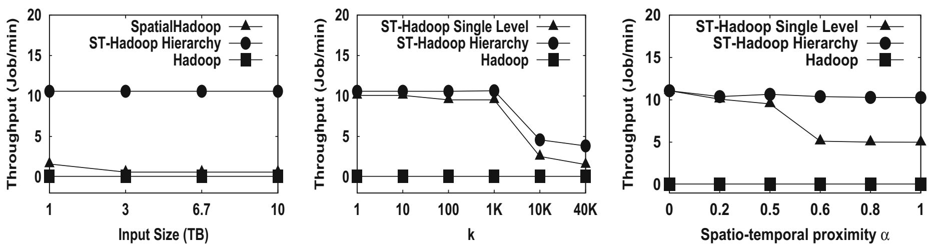
8.1 Spatiotemporal range query

In Fig. 14a, we increase the size of input from 1TB to 10TB, while measuring the job throughput. ST-Hadoop achieves more than two orders of magnitude higher throughput, due to the temporal load balancing of its spatio-temporal index. As for SpatialHadoop, it needs to scan more partitions, which explain why the throughput of SpatialHadoop decreases with the increase of data records in spatial space. Meanwhile, ST-Hadoop throughput remains stable as it processes only partition(s) that intersect with both space and time. Note that it is always the case that Hadoop needs to scan all HDFS blocks, which gives the worst throughput compared to SpatialHadoop and ST-Hadoop.

Figure 14b shows the effect of configuring the HDFS block size on the job throughput. ST-Hadoop manages to keep its performance within orders of magnitude higher throughput even with different block sizes. Extensive experiments are shown in Fig. 14c, analyzed how slicing ratio (α) can affect the performance of range queries. ST-Hadoop keeps its higher throughput around the default HDFS block size, as it maintains the load balance of data records in its two-layer indexing. As expected expanding the block size from its default value will reduce the performance on SpatialHadoop and ST-Hadoop, mainly because blocks will carry more data records.

8.2 *K*-nearest-neighbor queries (*k*NN)

Figures 15 give the performance of *k*NN query processing on Hadoop [36] and ST-Hadoop for 10 TB of twitter dataset. In experiments, 20 query locations are set at random points



(a) Input files (b) k (c) varying (α) of Ranking Function

Fig. 15 Spatio-temporal K-Nearest-Neighbor Query

(i.e., random points in both date and time) sampled from the input file, α is set to 0.4, the number of k is set to 100. Unless otherwise mentioned.

Figure 15a measures system throughput when increasing the input size from 1 TB to 10 TB. ST-Hadoop has one to two orders of magnitude higher throughput. Hadoop and SpatialHadoop performances decrease dramatically as they need to process the whole file while ST-Hadoop maintains its performance as it processes one partition regardless of the file size. Since SpatialHadoop is not aware of the temporal locality of the data, it needs to process multiple partitions to finds the k nearest neighbor in a specific day, and in a worst case it might end up processing all partitions. Hence, ST-Hadoop keeps its speedup at two orders of magnitude.

Figure 15b gives the effect of increasing k from 1 to 40K on 10 TB dataset. ST-Hadoop gives an order of magnitude performance with both single level index and optimized query plan that uses ST-Hadoop hierarchy index. ST-Hadoop achieves two orders of magnitude performance compared to Hadoop kNN implementation. ST-Hadoop efficiently handles spatio-temporal kNN operation. However, we notice that the job throughput decreases when k is more than eight thousand, where more partitions are required to be processed. ST-Hadoop is consistently better than Hadoop. While the performance with single level index tends to decrease with the increased number of the nearest neighbor k. In the meantime, the optimized query that uses the hierarchy index remains stable for a higher number of neighbors. However, at some point, it will decrease. This is expected as the number of selected partitions increase with the increased of the k number.

In Fig. 15c, shows how the job throughput affected by the value of α in the ranking function. The query point Q is fixed at random location on the first day of a month. The increase of α means that ST-Hadoop might need to process several partitions from different days and also nearby partitions within the same days to find the nearest neighbor. As the α value increases, the performance of ST-Hadoop stays at two orders of magnitude higher than Hadoop. Without having ST-Hadoop hierarchy index, the performance slightly decrease. This is expected as query cares more about spatial proximity, and thus, 30 partitions will need to be processed. The reason for the steady throughput by ST-Hadoop goes to the execution plan supplied by ST-Hadoop query optimizer. The query optimizer selects a single partition that overlap with the query point Q, which best fit to cover the whole temporal range. When $\alpha = 0$ the query optimizer overlaps Q with a single partition from the highest granularity (e.g, daily level). If $\alpha = 1$, then the query optimizer selects a single partition from a lower granularity level(e.g., month). Meanwhile, if α is in between that two extremes, then the query optimizer selects a single partition that either extends over

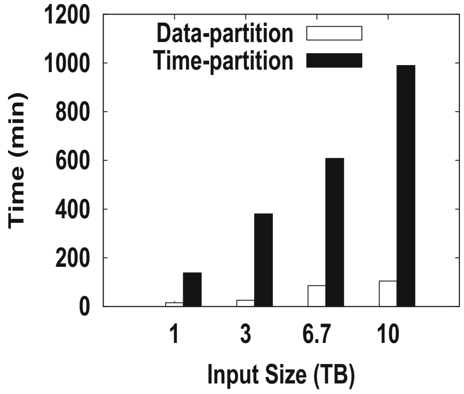


Fig. 16 Input Files

the whole temporal range or partially cover it based on the ranking score of the furthest *k* (Figs. 16 and 17).

8.3 Spatiotemporal join

Figure 18 gives the results of the spatio-temporal join experiments, where we compare our join algorithm for ST-Hadoop with MapReduce implementation of the spatial hash join algorithm [19]. Typically, in this join algorithm we perform the following query, "*find every pairs that are close within an Euclidean distance of* $1mile$ *and a temporal distance of* $2days$", this join query is executed on both ST-Hadoop and Hadoop and the response times are compared. The y-axis in the figure represents the total processing time, while the x-axis represents the join query on numbers of days×days in ascending order. With the increase of joining number of days, the performance of ST-Hadoops join increases, because it needs to join more indexes from the temporal hierarchy. In general, ST-Hadoop gives the best results as ST-Hadoop index replicates data in several layers, and thus ST-Hadoop significantly decreases the processing of non-overlapping partitions, as only partitions that overlap with both space and time are considered in the join algorithm. Meanwhile, the same joining algorithm without using ST-Hadoop index gives the worst performance for joining spatio-temporal data, mainly because the algorithm takes into its consideration all data records

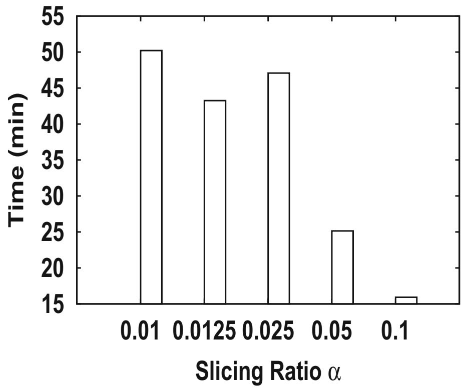
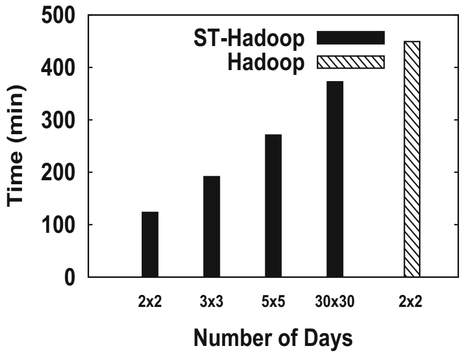


Fig. 17 Data-partition

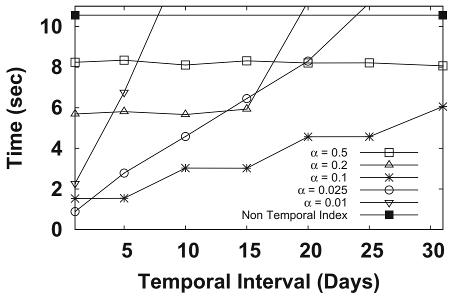
Fig. 18 Spatio-temporal Join



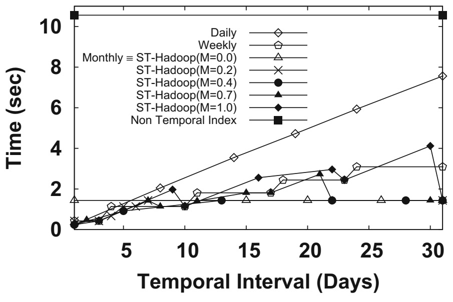
from one dataset. However, ST-Hadoop only joins the indexes that are within the temporal range, which significantly outperforms the join algorithm with double to triple performance.

8.4 Query optimizer

Experiments in Fig. 19 examines the performance of the temporal hierarchy index in ST-Hadoop using both slicing techniques. We evaluate different granularities of time-partition slicing (e.g., daily, weekly, and monthly) with various data-partition slicing ratio. In these two figures, we fix the spatial query range and increase the temporal range from 1 day to 31 days, while measuring the total running time. As shown in the Figs. 19a and b, ST-Hadoop utilizes its temporal hierarchy index to achieve the best performance as it mixes and matches the partitions from different levels to minimize the running time, as described in Section 7. ST-Hadoop provides good performance for both small and large query intervals as it selects partitions from any level. When the query interval is very narrow, it uses only the lowest level (e.g., daily level), but as the query interval expand it starts to process the above level. The value of the parameter $\mathcal{M}$ controls when it starts to process the next level. At $\mathcal{M} = 0$, it always selects the up level, e.g., monthly. If $\mathcal{M}$ increases, it starts to match with lower levels in the hierarchy index to achieve better performance. At the extreme value of $\mathcal{M} = 1$, the algorithm only matches partitions that are completely contained in the query interval, e.g., at 18 days it matches two weeks and four days while at 30 days it matches the whole month. The best choice of $\mathcal{M}$ value in this experiment is $\mathcal{M} = 0.4$ which means it only selects partitions that are at least 40% covered by the temporal query interval.

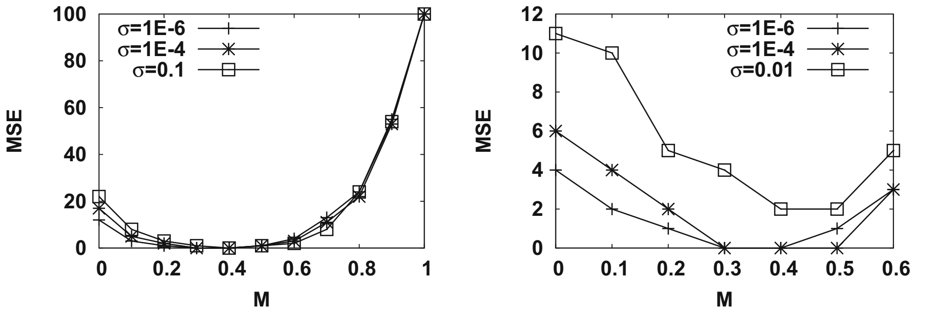


(a) Data-partition Slicing



(b) Time-partition Slicing

Fig. 19 Spatio-temporal Range Query Interval Window

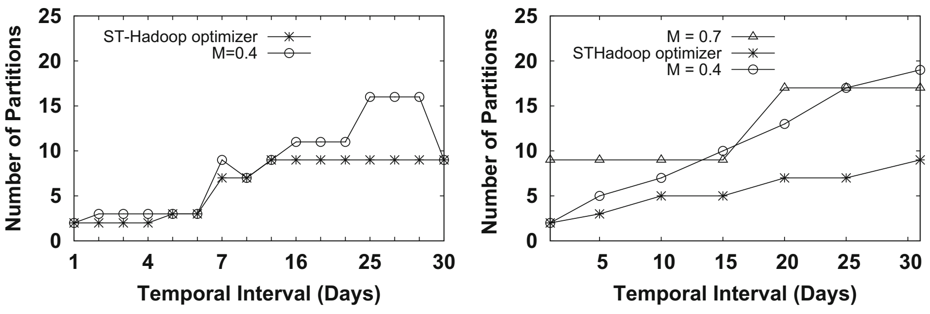


(a) Tuning of $\mathcal{M}$ for query intervals from 1 to 30 days (b) Tuning of $\mathcal{M}$ for query intervals from 1 to 400 days

Fig. 20 The effect of the spatio-temporal query ranges on the best value of $\mathcal{M}$

In Fig. 20 we study the effect of the spatio-temporal query range (σ) on the choice of $\mathcal{M}$. To measure the quality of $\mathcal{M}$, we define the best running time for a query Q as the minimum of all running times for all values of $\mathcal{M} \in [0, 1]$. Then, we determine the quality of a specific value of $\mathcal{M}$ on a query workload as the mean squared error (MSE) between the running time at this value of $\mathcal{M}$ and the best running time. This means, if a value of $\mathcal{M}$ always provides the best value, it will yield a quality measure of zero. As this value increases, it indicates a poor quality as the running times deviates from the best running time. In Fig. 20a, We repeat the experiment with three values of spatial query ranges $\sigma \in \{1E-6, 1E-4, 0.1\}$. As shown in the figure, $\mathcal{M} = 0.4$ provides the best performance for all the experimented spatial ranges. This is expected as $\mathcal{M}$ is only used to select temporal partitions while the spatial range (σ) is used to perform the spatial query inside each of the selected partitions. Figure 20b, shows the quality measures with a workload of 71 queries with time intervals that range from 1 day to 421 days. This experiment also provides a very similar result where the best choice value of $\mathcal{M}$ is around 0.4.

In Fig. 21 we evaluate ST-Hadoop greedy algorithm implemented in the new query optimizer with the heuristic approach of $\mathcal{M}$, on both slicing techniques supported in ST-Hadoop. Certainly, the best choice of $\mathcal{M}$ is at least as far ahead as the optimal, but it is not optimal. In the heuristic approach, a partition is selected only if its *coverage ratio* is above a



(a) Selected HDFS blocks in Time-partition Slicing (b) Selected HDFS blocks in Data-partition Slicing

Fig. 21 ST-Hadoop Greedy query optimizer VS heuristic $\mathcal{M}$

specific threshold $\mathcal{M}$, which is around 0.4. Meanwhile, in our new ST-Hadoop implementation we compute the exact number of partitions that need to be processed by employing a greedy algorithm that finds the minimum. ST-Hadoop employs a top-down approach that starts with the top level and selects partitions that cover query interval T, If the query interval T is not covered at that granularity, then the algorithm continues to the next level. ST-Hadoop finds the local optimal from each granularity until we reach the global optimal, i.e., the minimum number of partitions that covers query interval.

Figure 21a, compares the number of selected partitions between $\mathcal{M}$ and the greedy algorithm. The input files indexed and sliced through Time-partition technique, i.e., daily, weekly, monthly levels. We fix the spatial range query and increase the temporal range from 1 day to 31 days. In these experiments, we eliminate other $\mathcal{M}$ value, as experimentally we found that the best choice of $\mathcal{M}$ value is equal to (0.4). As shown in the figure, the greedy algorithm always beats $\mathcal{M}$. As expected the algorithm selects the global optimal, which is the minimal number of partitions that contain the final answer. In Fig. 21b, we repeated the same experiment with various data-partition slicing ratio. ST-Hadoop greedy algorithm provides the best performance for both small and large query intervals as it selects the minimum number of partitions from any level. When the query interval is very narrow, it uses only the lowest granularity level. As as the query interval expand query optimizer starts to process the above level.

8.5 Index construction

Figure 16 gives the total time for building the spatio-temporal index in ST-Hadoop. This is a one time job done for input files. In general, the figure shows excellent scalability of the index creation algorithm, where it builds its index using data-partition slicing for a 1TB file with more than 115 Million records in less than 15 minutes. The data-partition technique turns out to be the fastest as it contains fewer slices than time-partition. Meanwhile, the time-partition technique takes more time, mainly because the number of partitions are increased, and thus increases the time in physical writing phase.

In Fig. 17, we configure the temporal hierarchy indexing in ST-Hadoop to construct five levels of the two-layer indexing. The temporal indexing uses `Data-partition` slicing technique with different slicing ratio α. We evaluate the indexing time of each level individually. Because the input files are sliced into splits according to the slicing ratio, which directly effects on the number of partitions. In general with stretching the slicing ratio, the indexing time decreases, mainly because the number of partitions will be much less. However, note that in some cases the spatial distribution of the slice might produce more partitions as in shown with 0.25% ratio.

9 Conclusion

In this paper, we introduced ST-Hadoop [28] as a novel system that acknowledges the fact that space and time play a crucial role in query processing. ST-Hadoop is an extension of a Hadoop framework that injects spatio-temporal awareness inside SpatialHadoop layers. The key idea behind the performance gain of ST-Hadoop is its ability to load the data in Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. Hence, incoming spatio-temporal queries can have minimal data access to retrieve the query answer. ST-Hadoop is shipped with support for three fundamental spatio-temporal operations, namely, spatio-temporal range, top-k nearest neighbor, and join queries. However, ST-Hadoop is extensible to support a myriad of other spatio-temporal operations. We envision that ST-Hadoop will act as a research vehicle where developers, practitioners, and researchers worldwide, can either use directly or enrich the system by contributing their operations and analysis techniques.

References

1. https://about.twitter.com/company (2017)
2. Accumulo. https://accumulo.apache.org/
3. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J (2013) Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. In: VLDB
4. Al-Naami KM, Seker SE, Khan L (2014) GISQF: An Efficient Spatial Query Processing System. In: CLOUDCOM
5. Alarabi L, Mokbel MF, Musleh M (2017) St-hadoop: A mapreduce framework for spatio-temporal data. In: SSTD
6. Apache. Hadoop. http://hadoop.apache.org/
7. Apache. Spark. http://spark.apache.org/
8. Eldawy A, Mokbel MF (2014) Pigeon: A spatial mapreduce language. In: ICDE
9. Eldawy A, Mokbel MF (2015) SpatialHadoop: A MapReduce Framework for Spatial Data. In: ICDE
10. Eldawy A, Mokbel MF, Alharthi S, Alzaidy A, Tarek K, Ghani S (2015) SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data. In: ICDE
11. Erwig M, Schneider M (2002) Spatio-temporal predicates. In: TKDE
12. European XFEL: The Data Challenge, Sept. 2012. http://www.xfel.eu/news/2012/the_data_challenge
13. Fox AD, Eichelberger CN, Hughes JN, Lyon S (2013) Spatio-temporal indexing in non-relational distributed databases. In: BIGDATA
14. Fries S, Boden B, Stepien G, Seidl T (2014) Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In: ICDE
15. GeoWave. https://ngageoint.github.io/geowave/
16. Han W, Kim J, Lee BS, Tao Y, Rantzau R, Markl V (2009) Cost-based predictive spatiotemporal join
17. Kini A, Emanuele R Geotrellis: Adding Geospatial Capabilities to Spark, 2014. http://spark-summit.org/2014/talk/geotrellis-adding-geospatial-capabilities-to-spark
18. Li Z, Hu F, Schnase JL, Duffy DQ, Lee T, Bowen MK, Yang C (2016) A spatiotemporal indexing approach for efficient processing of big array-based climate data with mapreduce. IJGIS
19. Lo M-L, Ravishankar CV (1996) Spatial Hash-joins. In: SIGMODR
20. Lu J, Guting RH (2012) Parallel Secondo: Boosting Database Engines with Hadoop. In: ICPADS
21. Lu P, Chen G, Ooi BC, Vo HT, Wu S (2014) ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. PVLDB
22. Ma Q, Yang B, Qian W, Zhou A (2009) Query Processing of Massive Trajectory Data Based on MapReduce. In: CLOUDDB
23. Land Process Distributed Active Archive Center, Mar. 2015. https://lpdaac.usgs.gov/about
24. Data from NASA's Missions, Research, and Activities, 2016. http://www.nasa.gov/open/data.html
25. Nishimura S, Das S, Agrawal D, El Abbadi A $\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. DAPD
26. NYC Taxi and Limousine Commission, 2017. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
27. Pavlo A, Paulson E, Rasin A, Abadi D, DeWitt D, Madden S, Stonebraker M (2009) A Comparison of Approaches to Large-Scale Data Analysis. In: SIGMOD
28. ST-Hadoop website. http://st-hadoop.cs.umn.edu/
29. Stonebraker M, Brown P, Zhang D, Becla J (2013) SciDB: A Database Management System for Applications with Complex Analytics. Computing in Science and Engineering
30. Tan H, Luo W, Ni LM (2012) Clost: a hadoop-based storage system for big spatio-temporal data analytics. In: CIKM

31. Wang G, Salles M, Sowell B, Wang X, Cao T, Demers A, Gehrke J, White W (2010) Behavioral Simulations in MapReduce. PVLDB
32. Whitby MA, Fecher R, Bennight C (2017) Geowave: Utilizing distributed key-value stores for multidimensional data. In: Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD
33. Whitman RT, Park MB, Ambrose SA, Hoel EG (2014) Spatial Indexing and Analytics on Hadoop. In: SIGSPATIAL
34. Yokoyama T, Ishikawa Y, Suzuki Y (2012) Processing all k-nearest neighbor queries in hadoop. In: WAIM
35. Yu J, Wu J, Sarwat M (2015) GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data. In: SIGSPATIAL
36. Zhang S, Han J, Liu Z, Wang K, Feng S (2009) Spatial Queries Evaluation with MapReduce. In: GCC
37. Zhang X, Ai J, Wang Z, Lu J, Meng X (2009) An efficient multi-dimensional index for cloud data management. In: CIKM
38. Zhong Y, Zhu X, Fang J (2012) Elastic and Effective Spatio-Temporal Query Processing Scheme on Hadoop. In: BIGSPATIAL

Louai Alarabi is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Minnesota - Twin Cities. He received his M.Sc. from the same department in 2015. His research interests include big spatial and spatio-temporal data management. His current research focus is managing spatio-temporal data. His research work has been selected among best papers award in SSTD 2017. He has been selected a finalist for ACM SIGMOD Research competition 2017.

Mohamed F. Mokbel (Ph.D., Purdue University, MS, B.Sc., Alexandria University) is an Associate Professor in the Department of Computer Science and Engineering, University of Minnesota. His research interests include the interaction of GIS and location-based services with database systems and cloud computing. His research work has been recognized by five Best Paper Awards and by the NSF CAREER award. Mohamed was the program co-chair for the ACM SIGSPATIAL GIS conference from 2008 to 2010, IEEE MDM Conference 2011 and 2014, and the General Chair for SSTD 2011. He is an Asscoiate Editor for ACM TODS, ACM TSAS, VLDB journal, and GeoInformatica. Mohamed is a founding member of ACM SIGSPATIAL, and an elected Chair of ACM SIGSPATIAL 2014–C2017. For more information, please visit: www.cs.umn.edu/~mokbel.

Mashaal Musleh is a PhD student at the Computer Science and Engineering Department, University of Minnesota - Twin Cities, under the supervision of Prof. Mohammed Mokbel. Mashaal Obtained his BS degree in 2015 with first class honours from the Computer Engineering department at Umm Al-Qura University, Makkah, Saudi Arabia. His primary research interest is big data management system.