Chapter 6

Modeling and Indexing Spatiotemporal Trajectory Data in Non-Relational Databases

Berkay Aydin
Georgia State University, USA

Vijay Akkineni
Georgia State University, USA

Rafal A Angryk
Georgia State University, USA

ABSTRACT

With the ever-growing nature of spatiotemporal data, it is inevitable to use non-relational and distributed database systems for storing massive spatiotemporal datasets. In this chapter, the important aspects of non-relational (NoSQL) databases for storing large-scale spatiotemporal trajectory data are investigated. Mainly, two data storage schemata are proposed for storing trajectories, which are called traditional and partitioned data models. Additionally, spatiotemporal and non-spatiotemporal indexing structures are designed for efficiently retrieving data under different usage scenarios. The results of the experiments exhibit the advantages of utilizing data models and indexing structures for various query types.

INTRODUCTION

In recent years, the rapid advancements in satellite imagery technology, GPS enabled devices, location-based web services, and social networks caused a proliferation of massive spatiotemporal data sets (Nascimento, Pfoser, & Theodoridis, 2003). Many consumer-oriented applications from social networks (Facebook, Twitter, Swarm) to mobile services including routing (Google Maps, Apple Maps), taxi services (Uber) etc. consume and generate spatio-temporal location data (Quercia, Lathia, Calabrese, Lorenzo, & Crowcroft, 2010). Furthermore, there are many massive spatiotemporal data repositories generated
by scientific resources that are monitoring the moving objects. These include solar events (Schuh et al., 2013), animal migrations (Buchin, Dodge, & Speckmann, 2014), and meteorological phenomena (J. J. Wang et al., 2014). Most traditional relational database management systems provide efficient storage and retrieval schema for almost all types of data. However, usually, they are optimized for datasets of gigabytes of size and centralized processing. On the other hand, NoSQL databases, also known as non-relational databases, refer to a set of database systems that emphasize schema-free models, and ad hoc data organization. Many people are increasingly using these databases where scalability, high volume and fault tolerance of big spatiotemporal data are key deciding factors. NoSQL databases being an umbrella for several types of data stores such as key-value store, column store, document store, graph database and several other storage formats, there is no one-for-all spatiotemporal model used in non-relational databases; and appropriateness of a particular solution depends on the problems to be solved.

Relational database management systems such as PostgreSQL (with PostGIS), Oracle (Spatial and Graph) are designed to store, index and query data that represents geometric objects with spatial characteristics. However, because of the computationally expensive (both processing and storage-wise) spatial and spatiotemporal joins, the scalability of the relational databases are restricted. Many modern applications, including real time object tracking and spatiotemporal data analyses require massive amounts of data ingestion, storage, and query streaming. These tasks require a demand for horizontal scalability.

For solving these problems in traditional RDBMS settings, vertical scaling (increase in processing power and memory of an individual processing unit) is needed. In non-relational databases, horizontal scaling (increasing the number of computers/nodes in a distributed system) can be used for addressing such problems. For our work, we have used Apache Accumulo (Sawyer, O’Gwynn, Tran, & Yu, 2013), which is one of the popular column-based non-relational databases with notable features such as load balancing, horizontal scalability, automatic table partitioning (which will be presented in detail in Related Work section). Accumulo also provides custom server-side iterators that can be efficiently utilized when performing spatiotemporal operations needed in queries involving spatiotemporal predicates.

Specifically, in this work, we have approached the problem of storing massive trajectory-based spatiotemporal data in the context of non-relational databases. One part of the problem is the representation of a spatiotemporal trajectory to fit the underlying storage system. Before all else, the design of data models for storing spatiotemporal trajectories in key-value stores is presented. For comparison purposes, our first class of data models (traditional data model) mimics the traditional object-relational database organization. On the other hand, our second class of data models (partitioned data model) exploits the sorted nature of row identifiers in Accumulo database, and stores data using the identifiers of spatiotemporal partitions. For increasing the query performance of proposed data models for different scenarios, in-memory indexing structures are also designed. Further discussion of the indexing structures can be found in Types of Queries and Indexing Trajectories section.

The rest of this chapter is organized as follows. In Related Work section, related work on storing and indexing spatiotemporal trajectories in traditional (single machine) and distributed storage systems is presented. In Non-relational Databases section, background information on distributed non-relational databases with a focus on Accumulo database will be provided. In Modeling Trajectories in Non-relational Databases, our data models for storing trajectories are demonstrated. Next, the different types of queries and the indexing strategies for increasing the query performances are shown. Lastly, we present our experimental evaluation; and point future directions; then, conclude the paper.
RELATED WORK

R-tree (Guttman, 1984) is the traditional data structure that handles the two dimensional (spatial) and three dimensional (spatiotemporal) data. R-trees can be seen as the spatial extension of B-trees, and indexes the spatial dimension of data using the nested bounding rectangles (or cubes for three dimensional data). For many spatiotemporal indexing structures, R-trees or some versions of R-trees are used for handling spatial (and sometimes temporal) dimensions in spatiotemporal data. In the literature, many spatiotemporal indexing structures are presented. These include multi-version structures such as HR-tree, STR-tree, MV3R-tree, or trajectory-oriented structures such as TB-tree, Polar-tree, Chebyshev Polynomial Indexing and SETI (Mokbel, Ghanem, & Aref, 2003) (Nguyen-Dinh, Aref, & Mokbel, 2010).

For indexing spatiotemporal trajectory data, initially, three dimensional R-trees have been used by considering time dimension as the third dimension and storing the whole trajectories or the segments of the trajectories into rectangular cuboids, which are reflected in the leaf nodes of R-tree. Trajectory-bundle trees (TB-tree) are optimized R-trees that preserves the trajectories by storing the segments of each in distinct leaf nodes (Pfoser, Jensen, & Theodoridis, 2000). Scalable and Efficient Trajectory Index (SETI), partitions the spatial dimension into fixed grid cells, and uses an R-tree for indexing the temporal dimension (Chakka, Everspaugh, & Patel, 2003). Start/End timestamp B-tree (SEB-tree) follows a fixed spatial partitioning strategy; and, each spatial partition is indexed using the start and the end times of the segments of the trajectory (Song & Roussopoulos, 2003). Conceptually similar (fixed spatial partitioning and data-driven temporal index) structures include Multiple Time-Split B-tree (MTSB-tree) (Zhou, Zhang, Salzberg, Cooperman, & Kollios, 2005) and Compressed Start-End Tree (CSE-tree) (Wang, Zheng, Xie, & Ma, 2008).

Aforementioned indexing strategies primarily focus on indexing the spatiotemporal data (in particular trajectories). On the other hand, storage systems, such as TrajStore (Cudre-Mauroux, Wu, & Madden, 2010), have been suggested for large-scale spatiotemporal data in traditional settings. Recently, several approaches were introduced for using distributed systems to store the spatiotemporal data. Geopot, introduced by Lee and Liang, discusses a cloud-based geo-location data management system with local in-memory spatial indexing structures (2011). Fox, Eichelberger, Hughes, and Lyon proposed a storage methodology for storing spatiotemporal data in Accumulo database (2013). Recently, Ke et al. proposed a practical spatiotemporal database schema based on MongoDB (MongoDB is classified as a NoSQL database) and a hybrid indexing approach spatiotemporal trajectories using HBSTR-tree, which combines hash table, B*-tree, and spatiotemporal R-tree (2014).

NON-RELATIONAL DATABASES

Non-relational databases (also referred as NoSQL databases) are storage systems that provide simple application programming interfaces for data retrieval without enforcing the constraints of the relational data model. CAP Theorem coined by Brewer (2000) states that any distributed system cannot guarantee consistency, availability and partition tolerance simultaneously. Relational Databases usually focus on the consistency and the availability. On the other hand, NoSQL databases (including Accumulo, database chosen for this work) are often optimized for the partition tolerance and availability aspects. Recently, NoSQL databases have become popular due to their capability of handling big data, as they are able to provide horizontal scaling schemata and simple data access mechanisms.
The proliferation of spatiotemporal data stems from following reasons (Shekhar & Chawla, 2003): (1) update operations on trajectory datasets are performed very commonly in real life datasets; (2) volume of the spatiotemporal trajectory datasets tends to be very big because of the stored geometric representations, and (3) data analytics performed on trajectory data requires massive data transfers.

**Accumulo Database**

In this work, for storing trajectories, we have opted using Accumulo database, which is a top-level Apache project among others like Cassandra and HBase (Fox, Eichelberger, Hughes, & Lyon, 2013). Accumulo is a column family oriented database, that is inspired by Google’s *BigTable* database model, where stored are key-value pairs, lexicographically sorted based on their keys (Chang *et al.*, 2008). In BigTable design, keys are comprised of row identifier, column identifier, timestamp; while value fields contain byte arrays. Accumulo, built using BigTable’s design, includes cell-level security, and more importantly, server-side iterators. Note that Accumulo is a sparse, distributed, sorted and multi-dimensional key-value storage system that depends on Apache Hadoop Distributed File System (HDFS) for data storage and Apache Zookeeper for configuration (Sen, Farris, & Guerra, 2013).

The key components of the Accumulo’s architecture, shown in Figure 1, are master, tablet servers, garbage collector, logger, and monitor. The main function of the *master* is to monitor the cluster for the

![Figure 1. The key components of Accumulo database](image-url)
status of tablet servers, assign tablets (partition of tables) to tablet servers, and perform load balancing. The master also handles tablet server failures and recovery processes. Tablet server component is responsible for handling all the reads and writes for the tables. In a typical deployment, one tablet server is co-located with one HDFS datanode. Tablet server gets registered with Accumulo software by obtaining a lock from Zookeeper. Another task of tablet servers is handling of minor and major compactions. Minor compaction is the process of flushing the data stored in memory to sorted files stored on disk. Major compaction is merging these sorted files into a bigger file. Additional components (not shown in Figure 1) include: (1) garbage collector that deletes files, which are no longer used, from HDFS; (2) monitor that is used for monitoring key metrics of system resources used by Accumulo; (3) logger for tracing the system events.

Some features of Accumulo that cater well for spatiotemporal trajectory data are automatic table partitioning, load balancing, horizontal scalability, server side iterators, and failure recovery (Sen, Farris, & Guerra, 2013). One convenient feature of Accumulo key-value store is automatic table partitioning, where the tables are split after crossing a pre-configured threshold. Using this feature, tables can be stored across multiple tablet servers evenly, and the system can provide parallelized access to data. Load balancing provided by Accumulo spreads the workload of tablets to tablet servers evenly and ensures that no tablet server is overloaded. Load balancing is essential when implementing a distributed system for scalability purposes (e.g. avoiding hotspots in spatiotemporal data analysis by distributing the workload). Furthermore, horizontal scalability (also known as scaling out –adding more nodes to a system for increasing the workload capacity) can be achieved with Accumulo. In contrast to traditional object-relational databases used for spatiotemporal data in single machine settings, the scaling in Accumulo is cheaper because of the commodity hardware used by the distributed file system in the back end of Accumulo. Note that, traditional databases are usually scaled vertically (also known as scaling up –increasing the system resources such as memory or processing power). The main way of retrieving data from Accumulo is via the server-side iterators whose main function is to traverse over the data with optionally filtering or transforming data. Server side iterators, which can lead to big performance increases by offloading some of the computations to the tablet servers. The key strength of the functionality and data representations provided by Accumulo is the ability to store sparse multi-dimensional data. This makes it a good candidate for storing and manipulating large-scale spatiotemporal trajectory data.

It is important to note the generic key-value data model of Accumulo database. Accumulo does not only have simple keys for each value, but key fields are partitioned into smaller conceptual fields that are: Row ID, Column (Family, Qualifier, Visibility), Timestamp. Row ID is the main attribute of a key, and the rows in tables are sorted primarily using Row IDs. Each Row ID is unique for a particular row, therefore, provides faster accessing mechanisms. Column fields (Family, Qualifier, and Visibility) sorts the table after Row IDs. Column Family determines the locality groups while Column Qualifier provides uniqueness within a row. Column Visibility controls user access. Timestamp field controls the versioning.

While we selected using Accumulo, the models presented in this work can be applied to other NoSQL databases with minimal modifications. Some of the other popular NoSQL databases include HBase, Cassandra, MongoDB, CouchDB, Redis, Memcached. Redis and Memcached databases are in-memory databases; therefore, the storage space is limited to available memory. MongoDB and CouchDB, both classified as document stores (the records are structured as JSON documents), are less-commonly used for very large datasets. HBase and Cassandra, more similar to Accumulo, are considered as column-oriented databases, and are inspired by BigTable design. Cassandra is a highly available system while
HBase and Accumulo values consistency more (see CAP theorem – (Brewer, 2000)). Note that HBase and Accumulo are very similar systems built on top of HDFS, and our models can be mapped to HBase with minimal efforts.

MODELING TRAJECTORIES IN NON-RELATIONAL DATABASES

Formal Definition of Spatiotemporal Trajectory

Our definitions follow the conceptual spatiotemporal trajectory (with its geometric facet) modeling from Spaccapietra et al. (2008). We formally define the spatiotemporal trajectories and its building blocks used in our work as follows. Firstly, a time-geometry pair object, denoted as $tGeo$, is a composite object representing certain geometry at a particular timestamp. Each time-geometry pair consists a valid timestamp, meaning the timestamp value cannot be empty. On the other hand, the geometry object can take all types of geometries defined in OGC (OpenGIS Consortium) specification (Cox et al., 2002), including points, polygons, etc. According to OGC, an empty geometry is also allowed.

$tGeo_i := \langle t_i, Geometry_i \rangle$

A trajectory, identified by a unique identifier (denoted as $trajId$) is an ordered list of timestamp-geometry pair objects. This list signifies the geometric representations of a trajectory in a particular time interval.

$Trajectory := trajId, [tGeo_{start}, tGeo_{start+1}, \ldots, tGeo_{end}]$

Note that we consider a continuous trajectory whose geometric representations are valid (alive) between start (birth) and end (death) times of the trajectory. Nevertheless, it is possible to have empty geometric representations. Each trajectory can only have one geometric representation for a particular timestamp. It is possible to use geometry collections (such as multipoint or multipolygon) to represent complex geometries.

The start and end times of a trajectory are the minimum and maximum time values, respectively, in the time-geometry pair list. They are denoted as $t_{start}$ (start time) and $t_{end}$ (end time). The start and end times are represented as a pair, and will be referred to as lifespan of trajectory hereafter. Lifespan of a trajectory is denoted as $lifespan_{Traj}$:

$lifespan_{Traj} := (t_{start}, t_{end})$

Minimum bounding rectangle of a trajectory (represented as $MBR_{Traj}$) signifies the smallest enclosing rectangle (orthogonal to the coordinate system used) representing maximum extent of all geometries included in the entire time-geometry pair list of a trajectory. The minimum bounding rectangle of a trajectory can be found by finding the minimum bounding rectangle of the union of all geometries belonging to that trajectory.
The lifespan and minimum bounding rectangle of trajectory show the temporal and spatial extensions of trajectories. Note that these attributes are derived from original time-geometry pairs list, and they will be called metadata attributes hereafter. They will be stored for the purpose of efficient spatial and temporal filtering mechanisms.

Additionally, for many applications, a trajectory can be associated with many non-spatiotemporal attributes. In this work, the only non-spatiotemporal attribute considered is the unique identifier of trajectory.

**Traditional Data Models**

In traditional data models, the spatiotemporal trajectories are treated as regular objects in object-relational database settings. By regular objects, we mean the objects that are stored without their spatial or temporal properties. Within these models, we have designed two alternatives for placing the data in the non-relational databases. The first one will be called column-fragmented traditional model, in which the trajectories are fragmented into their time-geometry pairs, where each geometry is stored using a different column identifier.

In the second alternative, which will be called whole-trajectory traditional model, all of the time-geometry pairs of each trajectory are serialized (into byte arrays) and stored as a whole using the same key (as a row identifier). In addition to those for extending the spatiotemporal search capabilities of traditional data models, we propose to use metadata attributes for storing spatial and temporal characteristics of trajectories that are: minimum bounding rectangle and lifespan of trajectory.

**Column-Fragmented Traditional Data Model (CTDM)**

In the column-fragmented traditional model, the individual time-geometry pairs, which the trajectories are composed of, are separately stored using different key values. Figure 2 demonstrates the hierarchical decomposition of keys and values to be used. The unique trajectory identifier (trajId) is used as the row identifier (Row ID) field of the Accumulo key. On the other hand, the column identifiers are different for each geometry of a trajectory. As the time-geometry pairs are uniquely identified by their timestamp value, and the column qualifiers are used for uniqueness in Accumulo, it is preferable to set column qualifiers as timestamp values of time-geometry pair objects in a trajectory. Therefore, the column-fragmented model stores the each trajectory using same row identifier (which is trajectory identifier) along with different column qualifiers (which represents the unique timestamps in time-geometry pairs list). Note that, many fields (column family, column visibility, timestamp) are left empty (‘NULL’ is used in Figure 2 and Algorithm 1); and, they may be used for additional tasks such as security and data versioning. Essentially, in the column-fragmented traditional data model, each row in Accumulo stores a single trajectory; yet, each geometry in the list of time-geometry pairs is stored in a different column, where each geometry is identified by its corresponding timestamp value. Therefore, it is easier to access and modify a single time-geometry pair of a trajectory in this data model.
**Algorithm 1: Insertion algorithm for column-fragmented traditional data model**

**Input:**
- Trajectory object: `traj`
- Minimum bounding rectangle of `traj`: `mbr`
- Lifespan of `traj`: `lifespan`

```plaintext
function CTDM_Insert
    rowIdentifier <- traj.trajId
    metaCF <- "metadata"  // column family value for metadata attributes
    // column qualifier values are set to "MBR" and "lifespan"
    insert_meta(rowIdentifier, metaCF, "MBR", mbr)
    insert_meta(rowIdentifier, metaCF, "lifespan", lifespan)
    for each tGeo in traj.timeGeoList
        columnFamily <- NULL
        columnQualifier <- tGeo.t
        value <- serialize(tGeo.Geometry)
        insert(rowIdentifier, columnFamily, columnQualifier, value)
    endfor
end function
```

Furthermore, we project and recommend metadata attributes of each trajectory be stored in separate tables from the actual trajectory tables for faster access. However, the metadata attributes can also be stored using a separate column family value as in the insertion algorithm shown in Algorithm 1—the string, “metadata”, is used for the column family field (as a name of the column) for MBR and lifespan of trajectories.
Whole-Trajectory Traditional Data Model (WTDM)

In the whole-trajectory traditional model, the time-geometry pairs of a trajectory are stored as a whole, within a serialized binary object. The Figure 3 demonstrates the hierarchical decomposition of keys and values. Similar to column-fragmented alternative, row identifiers are set to be the trajectory identifiers. However, the entire list of time-geometry pairs of a trajectory is stored as a serialized byte array. The column identifiers (column family, column qualifier, column visibility) are left empty (‘NULL’ is used in Figure 3 and Algorithm 2) in this model. Similar to column-fragmented model, the metadata fields for minimum bounding rectangle and lifespan of the trajectory are also stored for efficient access.

This model provides faster access to the whole trajectory compared to column-fragmented traditional data model. Note that in the column-fragmented alternative, the geometries are fragmented into columns, whereas in the whole-trajectory traditional data model, all of the time-geometry pairs are stored together in a single serialized byte array. In the whole-trajectory traditional data model, it can be more difficult to modify the individual geometry objects.

Insertion and Search Algorithms

Algorithm 2: Insertion algorithm for whole-trajectory traditional data model

Input:
Trajectory object: traj
Minimum bounding rectangle of traj: mbr
Lifespan of traj: lifespan

function WTDM_Insert

rowIdentifier <- traj.trajId
metaCF <- “metadata” //column family value for metadata attributes
//column qualifier values are set to “MBR” and “lifespan”
insert_meta(rowIdentifier, metaCF, “MBR”, mbr)

Figure 3. Hierarchical decomposition of key fields in whole-trajectory traditional data model
insert_meta(rowIdentifier, metaCF, "lifespan", lifespan)
value <- serialize(traj.timeGeoList)
columnFamily <- NULL
columnQualifier <- NULL
insert(rowIdentifier, columnFamily, columnQualifier, value)
end function

The *insertion* and generic *spatiotemporal search* (spatiotemporal range or window query) algorithms for the column-fragmented and whole-trajectory traditional data model are presented in Algorithms 1 through 3. The insertion Algorithms 1 and 2, insert the metadata attributes (note column family field is set to “metadata” – as a string); and, store the well-known binary (WKB) formatted geometric representation of geometry fields of time-geometry pairs in trajectory as demonstrated in the respective algorithms. The generic spatiotemporal search algorithm (Algorithm 3) initially applies a filter-and-refine step using above-mentioned metadata attributes with spatial and temporal predicates (See *metadata_search* function in Algorithm 3). For example, suppose the given are a spatiotemporal window defined over a spatial rectangle and a temporal interval; as well as *ends-before* temporal predicate and *overlaps* spatial predicate. Metadata search (*metadata_search*) procedure initially filters the trajectories using temporal predicate ends-before by selecting trajectories whose end times are less than the interval’s end time. Later, the filtered trajectories’ minimum bounding rectangles are examined whether they intersect with queried spatial rectangle for correctness and completeness of spatial overlap predicate. Lastly, the resulting trajectory identifiers are returned to spatiotemporal search for further inspection (by firstly checking temporal, then spatial predicates with the given query) of the given spatiotemporal window (spatial rectangle and time interval), as the overlap of minimum bounding rectangles does not guarantee the true spatiotemporal overlap.

Algorithm 3: Generic spatiotemporal search algorithm for traditional data models

function search_TDM
    results <- []
    refinedTrajectoryIds <- *metadata_search* (qp, sp, tp)
    for each trajId in refinedTrajectoryIds
        trajectory <- *getTrajectoryFromDatabase* (trajId)
        if (*checkTemporalPredicate* (trajectory, qp, tp))
            if (*checkSpatialPredicate* (trajectory, qp, sp))
                results.add(trajectory)
        end if
    end for
    return results
end function

Input:
A query predicate which consists of spatial and temporal bounds: \(qp\)
Spatial predicate showing condition: \(sp\)
Temporal predicate showing condition: \(tp\)
Updates on Traditional Data Models

The updates on spatiotemporal trajectories for traditional data models can be inspected in two parts: (1) updates on non-spatiotemporal attributes, and (2) updates on spatiotemporal attributes, namely the time-geometry pairs list. For both of the traditional data models, updates on non-spatiotemporal fields (we only consider the unique trajectory identifier as a non-spatiotemporal attribute) can be performed by finding the Row ID (which stores trajectory identifier) and simply updating it to desired value.

The updates on spatiotemporal attributes, on the other hand, are more complicated. For column-fragmented traditional data model, a particular update on the time-geometry pairs of a trajectory requires either addition, deletion, or modification of geometries and timestamps. While handling timestamps is intrinsically easier, updates on geometries require overriding the entire geometry or multiple geometries. Another challenge of updating spatiotemporal attributes is manipulating of metadata attributes. The updates on the lifespans of trajectories are easy and straightforward, while the updates on geometries are more difficult. When updating geometries, it is necessary to fetch the geometries of entire trajectory and re-calculate the MBR, in the event of an update that includes either modification or deletion of geometries, as it can reduce the size of MBR. When geometry updates only include addition of a geometry (or a list of geometries), it is only necessary to fetch the MBR and re-calculate the MBR using the MBR of the added geometry (or geometries). For whole-trajectory data model, it is necessary to fetch entire geometry for any update and re-write it back, as the time-geometry pairs of trajectory are stored as a serialized object. Therefore, we can argue that whole-trajectory data model may not be well-suited for frequently updated spatiotemporal trajectory data.

Algorithm 4: Trajectory cell partitioning algorithm for traditional data models

**Input:**
Trajectory object: traj
Coordinates of MBR of traj: \((\text{minX}, \text{maxX}, \text{minY}, \text{maxY})\)
Start and end times of traj: \((\text{ts}, \text{te})\)
Partitioning thresholds of space and time dimensions: \(\Delta X, \Delta Y, \Delta T\)

**function** partition

\[
\text{cells} \leftarrow []
\]

\[
\text{for timeStep from ts}/\Delta T \text{ to te}/\Delta T
\]

\[
\text{for } \text{xStep from minX}/\Delta X \text{ to maxX}/\Delta X
\]

\[
\text{for } \text{yStep from minY}/\Delta Y \text{ to maxY}/\Delta Y
\]

\[
\text{cell}_i \leftarrow \text{create3D_cell}(\text{timestep}, \text{xStep}, \text{yStep})
\]

\[
\text{if } (\text{traj.intersects}(\text{cell}_i)
\]

\[
\text{cells.add(\text{cell}_i)}
\]

\[
\text{end if}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{return cells}
\]

end function
Partitioned Data Models

While traditional data models use trajectory identifiers as the row identifiers in key fields, partitioned data models follow an approach in which the spatiotemporal characteristics of the trajectories are taken into account. A space-driven partitioning strategy is followed for the partitioned data models. While dynamic trajectory splitting/partitioning techniques are present (Rasetic, Sander, Elding, & Nascimento, 2005), a fixed spatiotemporal partitioning strategy has been used for our models. Note that, we are aiming to make use of the ordered nature of row identifiers in Accumulo by creating an artificial and fixed row identifier carrying the implicit spatiotemporal semantics extracted from the time-geometry pairs of trajectories.

The pseudo code for partitioning algorithm can be seen in Algorithm 4. Partitioning algorithm takes user-defined partitioning thresholds of $\Delta X$, $\Delta Y$, and $\Delta T$, which are utilized for creating the three-dimensional spatiotemporal cells (or voxels). Cells are uniquely identified with a partition identifier, composed from the three spatiotemporal dimensions. Temporal dimension is used as the leading dimension for the partition identifier, as it offers a natural ordering (unlike spatial dimensions that might need space-filling curves to create artificial ordering). Partitioning algorithm computes the set of three-dimensional cells that the time-geometry pairs of trajectory span through.

Next, we suggest two partitioned data models: (1) Whole-trajectory partitioned model, and (2) Segmented-trajectory partitioned model. These models will be explained in the following sections.

Whole-Trajectory Partitioned Data Model (WPDM)

Similar to whole trajectory traditional data model, the time-geometry pairs list of a trajectory is stored as a whole after serializing it into a byte array (namely, a binary object). However, the row identifier is the partition cell identifier, and the column qualifier is the trajectory identifier. For trajectories spanning through multiple partitioning cells, the same data (serialized list of time-geometry pairs) is duplicated for each cell. The decomposition of keys and values is shown in Figure 4. In Algorithm 5, the pseudocode for inserting a trajectory to database using whole-trajectory partitioned data model is demonstrated. Insertion algorithm initially determines the partitions of the trajectory; then, inserts the serialized time-geometry pairs list into the database using partition identifiers as the Row IDs.

The whole-trajectory partitioned model offers a crude but efficient accessing mechanism especially for spatiotemporal range queries, using the partition cell identifiers as row identifiers. However, it is important to note that because of the fixed partitioning schema, a trajectory can intersect with many partition cells, and the data may be replicated many times, causing the space overhead. Therefore, it should be expected that the whole-trajectory partitioned data model requires more space than other models presented in this chapter. (See the Experiments section for more detailed results.)

Segmented-Trajectory Partitioned Data Model (SPDM)

Algorithm 5: Insertion algorithm for whole-trajectory partitioned data model

Input:
Trajectory object: $\text{traj}$
Coordinates of MBR of traj: $(\text{minX, maxX, minY, maxY})$
Start and end times of traj: $(ts, te)$
Partitioning thresholds of space and time dimensions: $\Delta X$, $\Delta Y$, $\Delta T$
In the segmented-trajectory partitioned data model, to decrease the amount of data duplications, the trajectory is firstly separated into trajectory segments if it spans through multiple spatiotemporal cells. Each trajectory segment has an associated trajectory identifier as well as a segment identifier. Segment identifier is essentially the same with the partition cell identifier (See Algorithm 4 for the determination of partition cell identifiers). A trajectory segment is defined as follows – “A trajectory segment contains a sublist of time-geometry pairs of its associated trajectory, where each of the time-geometry pairs belongs to one and only one partition cell”. Given these, a trajectory segment has a trajectory identifier and a segment identifier representing the partition cell it belongs to. Each trajectory may only have one and only one trajectory segment for one partition cell. However, a time-geometry pair may belong to one or more partition cells.

```
function WTPM_Insert
    rowIds <- partition(traj, (minX, maxX, minY, maxY), (ts, te), ΔX, ΔY, ΔT)
    columnFamily <- NULL
    for each cellId in rowIds
        columnQualifier <- traj.trajId
        value <- serialize(traj.timeGeoList)
        insert(cellId, columnFamily, columnQualifier, value)
    endfor
end function
```
The hierarchical decomposition of the segmented-trajectory partitioned data model is shown in Figure 5, and is very similar to the whole-trajectory partitioned data model. The insertion algorithm can be seen in Algorithm 6. Note that \texttt{partition\_segment} function in Algorithm 6 creates the trajectory segments for a particular trajectory. The segment identifier (which is identical with the partition cell identifier) is set to be the row identifier, while trajectory identifier is the column identifier. The difference between two models is in the value fields. For the whole-trajectory partitioned model the entire trajectory is serialized and stored, on the other hand, for the segmented-trajectory partitioned model, stored are the parts (which is called segments) of trajectories that are spanning in that particular cell identified by segment (cell) identifier. Additionally, the list of segments (partition cells) of a particular trajectory is also stored to enable access to the other segments of a particular trajectory.

Algorithm 6: Insertion algorithm for segmented-trajectory partitioned data model

\textbf{Input:}
Trajectory object: \texttt{traj}
Coordinates of MBR of \texttt{traj}: (\texttt{minX}, \texttt{maxX}, \texttt{minY}, \texttt{maxY})
Start and end times of \texttt{traj}: (\texttt{ts}, \texttt{te})
Partitioning thresholds of space and time dimensions: \(\Delta X\), \(\Delta Y\), \(\Delta T\)

\begin{verbatim}
function STPM_Insert
    <rowIds, segments> <- \texttt{partition\_segment}(\texttt{traj},
              (\texttt{minX}, \texttt{maxX}, \texttt{minY}, \texttt{maxY}),
              (\texttt{ts}, \texttt{te}), \(\Delta X\), \(\Delta Y\), \(\Delta T\))
    //\texttt{partition\_segment} partitions the trajectory into trajectory segments
    //and returns the cellIds and trajectory segments as a map
    columnFamily <- NULL
    for each (cellId, segment) in <rowIds, segments>
        columnQualifier <- \texttt{traj}\texttt{.trajId}
\end{verbatim}

Figure 5. Hierarchical decomposition of key fields in segmented-trajectory partitioned data model
The segmented-trajectory partitioned data model requires the extra storage of partition cell identifiers (or segment identifiers). However, in a particular row, identified by the partition cell identifier, the stored are the time-geometry pairs of the trajectory segments spanning in that particular cell. Therefore, it requires less space than whole-trajectory partitioned data model (see Experiments section), while providing faster access mechanisms compared to traditional data models for spatiotemporal queries.

Insertion, Spatiotemporal Search and Updates on Partitioned Data Models

Insertion algorithms for partitioned data models are shown in Algorithm 5 and 6. For both of the partitioned data models, insertion starts with the identification of partition cells. For each partition cell, either all of the time-geometry pairs (whole-trajectory partitioned data model), or the segments of time-geometry pairs (segmented-trajectory partitioned data model) are inserted as serialized binary objects. Note that for the segmented-trajectory partitioned data model, an extra step for segment partitioning is necessary.

The spatiotemporal search for partitioned data models (similar to the one described in Algorithm 3 for traditional data models) is intuitive. The generic spatiotemporal search initially requires the identification of partition cells the spatial and temporal predicates span through. Next, for all the identified partition cells, either whole trajectories or the segments of trajectories are returned. Later, those trajectories are further refined by checking the spatiotemporal predicates. Note that, duplicates should be eliminated when initially returning the results from database.

The updates for partitioned data models can be divided into two parts, the same as traditional data models. The updates on non-spatiotemporal attributes of a trajectory require brute-force search on unique trajectory identifiers, and later modification of the trajectory (or the segments of the trajectory).

The updates on spatiotemporal attributes are more complicated. For an update on a trajectory stored using whole-trajectory partitioned data model, the time-geometry pairs list of the trajectory is fetched from the database; and, the temporary update on the list is performed. Later, it is checked whether the partition cells of trajectory are changed. If they are not changed, the serialized time-geometry pairs list object of the trajectory is overwritten for all the partitions it spans through. If the updated partition cells are different from the previous version, the partition cells of trajectory that are valid for both versions are updated by overwriting the time-geometry pairs list. If applicable, the serialized time-geometry pairs list is inserted into new cells that trajectory spans through; or, the trajectory is deleted from the partition cells that it does not span through in the updated version.

For an update on a trajectory stored using segmented-trajectory partitioned data model, the updates are locally performed on each segment, when applicable. Similar to the whole-partitioned data model, the all of the trajectory segments are fetched, and updates are temporarily performed on the trajectory. Later, the new list of segments are created; and the database is updated accordingly by either modifying the segment in a particular cell, inserting a new segment in a different partition cell, or removing the segment of the trajectory from a partition cell.
In our system, we want to support the following query types:

- **Spatiotemporal Queries:** The primary goal of our work is to create data models that can answer queries with spatial or temporal or spatiotemporal predicates. By spatiotemporal queries, we mean queries that return entries in the database using the spatial or temporal aspects of our data. The specific use cases we have designed for our data models are following.
  - **Time range query:** Given a time range representing a time interval with two timestamps $t_{\text{start}}$ and $t_{\text{end}}$ ($t_{\text{start}} < t_{\text{end}}$), time range queries return trajectories having a temporal Boolean attribute in relation to the given timestamp. For example, for the query “Find trajectories that were alive in interval from ‘September 10\textsuperscript{th}, 2014 05:00:00’ to ‘September 11\textsuperscript{th}, 2014 05:00:00’”, returns trajectories whose lifetime overlaps with given interval from ‘September 10\textsuperscript{th}, 2014 5:00 AM’ to ‘September 11\textsuperscript{th}, 2014 5:00 AM’. Considered temporal predicates for trajectories include starts-before, starts-after, ends-before, ends-after, overlaps, and disjoint.
  - **Timestamp query:** Given a timestamp representing a unique moment, timestamp queries search for trajectories having a temporal Boolean attribute in relation to the given timestamp. For instance, provided a timestamp query translated to natural language as “Find trajectories that ends-before ‘September 10\textsuperscript{th}, 2014 05:00:00’”, trajectories whose end times are before the given timestamp ‘September 10\textsuperscript{th}, 2014 05:00:00’ is included in query results. Considered temporal predicates for trajectories include starts-before, starts-after, ends-before, ends-after, overlaps, and disjoint.
  - **Purely spatial query:** Given a spatial object represented as a geometry (point, line, polygon etc.), purely spatial queries search for trajectories satisfying a spatial Boolean relation to union of all the geometries in the list of time-geometry pairs of trajectories. The considered spatial predicates include geometrical relations such as contains, covered-by, covers, equal, inside, overlaps, disjoint, touches, crosses. An example query could be searching for the trajectories that covers the given spatial point ‘POINT (1.0, 2.0)’. Purely spatial queries could be separated into subclasses (e.g. spatial point, spatial polygon, spatial line, spatial multipoint etc.). However, for simplicity, they are investigated under the same kind of queries. Note that spatial predicates are not available for all types of geometries. For example, a spatial point type query cannot cover any trajectories, but may be covered-by a trajectory. On the other hand, a spatial polygon type query can cover and be covered-by trajectories.
  - **Spatiotemporal point query:** Given a spatial object represented as a geometry and a time-stamp (as in timestamp queries), spatiotemporal point queries return the trajectories that are satisfying both temporal and spatial Boolean relationships. The spatiotemporal predicates include the ones in timestamp queries aggregated to purely spatial queries.
  - **Spatiotemporal window query:** Given a set of spatial objects represented as a list of geometries and a time range, spatiotemporal window queries return the trajectories that are satisfying Boolean temporal and spatial relationships listed in (time range and purely spatial queries). Spatiotemporal window queries are common for spatiotemporal join operations. For example, finding trajectories that are overlapping in both spatial and temporal dimensions can be done using spatiotemporal window queries.
Non-spatiotemporal Queries: While spatiotemporal queries are our major part of the work, queries that are used for searching non-spatiotemporal attributes are also included. As the only non-spatiotemporal attribute of trajectories in our work is trajectory identifier, we only include search based on the trajectory identifier. The supported queries are as follows.

- **Exact query**: Given a specific trajectory identifier, the exact queries search for the trajectory specified by this particular trajectory identifier.
- **Range query**: Similar to exact queries, given a range of trajectory identifiers (specified by two trajectory identifiers), range queries search for trajectories whose trajectory identifiers fall in that particular range.

As mentioned earlier, the traditional models and the partitioned models have their weaknesses and strengths for particular types of queries. Traditional data models provide a better searching schema for non-spatiotemporal queries, while partitioned data models offer more promising searching structures for spatiotemporal queries. However, given the structure of our non-relational database, traditional data models does not suit well for spatiotemporal queries, while partitioned data models is not very appropriate for efficiently answering non-spatiotemporal queries. In order to eradicate these weaknesses, we suggest indexing methodologies for traditional and partitioned data models.

**Spatiotemporal Indexing for Traditional Data Models**

Scalable and Efficient Trajectory Index (SETI) was introduced by Chakka, Everspaugh, & Patel (2003). SETI is specifically designed for indexing trajectories and the intended use for this indexing structure is historical and persistent spatiotemporal data. For handling the spatial dimensions of trajectories, SETI uses a fixed grid. Each grid cell has an associated R-tree for handling temporal dimension.

For indexing trajectories in non-relational databases, we consider a version of SETI, which we will call Grid-mapped Interval Trees (G-IT) that uses interval trees for indexing temporal dimension and a fixed grid for indexing spatial dimensions. An illustration of the trajectory insertion to G-IT can be seen in Figure 6. The fixed grid file is formed using two parameters that are $\Delta X$ and $\Delta Y$, which would be used as the step sizes of spatial cells. The geometric representation of trajectories may span through more than one cells as the cells are fixed. For handling the trajectories that are spanning through more than one spatial cells, a duplication strategy is followed (Aydin *et al.*, 2014). Nevertheless, it is important to create the fixed spatial cells in such a way that there will be the minimal amount of duplications (in other words, the minimum amount of trajectories that are crossing more than one spatial cells). For minimizing the number of duplications, initial knowledge of movement and directional patterns of trajectories plays a remarkable role. Such knowledge can help determining the step sizes ($\Delta X$ and $\Delta Y$) of spatial cells, and reduce the number of duplications. For instance, in solar event based datasets (Schuh *et al.*, 2013), the movement is from left to right, selecting a larger horizontal step size and smaller vertical step size can decrease the amount of duplications.

Each cell in the fixed grid file is mapped to an interval tree handling the temporal dimension of the trajectories. Interval trees are essentially 1-dimensional R-trees that are usually preferred specifically for indexing interval-based time dimension (de Berg, van Krefeld, Overmars, & Schwarzkopf, 2000). Stored in the leaf cells of the interval trees are the trajectory identifiers. Note that, as we are designing the index to be an in-memory structure, we want to keep the leaf nodes as small as possible to eradicate the possible memory problems.
For the dynamic updates on the G-IT index, we followed a strategy similar to the update strategy presented for SETI (Chakka, Everspaugh, & Patel, 2003). The update procedure on G-IT starts with identifying the spatial cells that trajectory spans through which will be affected by a particular update. Note that the identification of the affected spatial cells requires fetching all the information about the trajectory stored in the index. In the case of an update that enhances the trajectory into new spatial cells, the trajectory identifiers are inserted to spatial cells on corresponding time intervals. For all other updates, the operation is performed by re-insertion of the trajectory to the index using modified time intervals.

G-IT index can be particularly well-suited for aforementioned spatiotemporal queries. For purely temporal query types (timestamp, time range queries) each interval tree in the index must be visited. The search in each interval tree requires logarithmic time (as the interval trees are R-tree based) with respect to the number of trajectories being stored in the respective interval tree. On the other hand, purely spatial queries require the search in the grid file, which requires constant time. The spatiotemporal queries in G-IT can be seen as a two-step filtering mechanism: spatial and temporal filtering. In a spatiotemporal query, firstly, the spatial predicate and the spatial portion of the query filters the cells that are not related to the query. Later, from the cells that are filtered, the temporal predicate and temporal portion of the query returns the actual results after eliminating the duplicates.
Inverted Index for Partitioned Data Models

Traditionally, the inverted index is used for text retrieval where each index entry (generally a word) points to the documents where the queried word occurs (Baeza-Yates & Ribeiro-Neto, 1999). The partitioned data models have better spatiotemporal querying characteristics as they encode the spatiotemporal semantics of trajectories in the keys. However, a direct non-spatiotemporal query to the non-relational database may result in unpredictable time requirements, simply because Accumulo is a column data store with no secondary indexes. For example, searching for a trajectory identifier that is not present in the database simply requires checking each entry in the database. In order to solve this unpredictable querying performance problem for partitioned data models, we suggest using a partition inverted index.

In partition inverted index, we treat a non-spatiotemporal attribute of a particular trajectory as the index entry (or a word in traditional text-retrieval oriented inverted index) and the partition cells as the documents. Figure 7 demonstrates an overview of partition inverted index to be used for partitioned data models. Simply, in the actual database, the indexed keys map the partition cells to the trajectories’ trajectory identifiers and time-geometry pairs list; on the other hand, in the partition inverted index, the trajectory identifiers are mapped to the partition cell identifiers. A sorted multi-valued map structure is used for building the inverted index structure. Note that the map structure is based on a balanced tree implementation that have a logarithmic insertion and search time requirement. Therefore, each trajectory identifier that is mapped to one or more cells is stored in sorted fashion in the index.

The partition inverted index can be used for answering non-spatiotemporal queries listed earlier more efficiently. For exact queries, the partition index performs an exact value search on the index and returned values are later used for fetching the actual trajectories from the database. Specified by two trajectory identifiers, range queries return a set of trajectory identifiers. When performing range queries, the sorted nature of the partition inverted index is utilized. As balanced trees are used, the non-spatiotemporal range query requires logarithmic time.

Figure 7. Partition inverted index

<table>
<thead>
<tr>
<th>Trajectories Stored in Database</th>
<th>Partition Inverted Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Row</strong></td>
<td><strong>Index Key</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Index Entry</strong></td>
</tr>
<tr>
<td>$\text{Partition}_1$</td>
<td>$\text{Traj}_3, \text{Traj}_4$</td>
</tr>
<tr>
<td>$\text{Partition}_2$</td>
<td>$\text{Traj}_1$</td>
</tr>
<tr>
<td>$\text{Partition}_3$</td>
<td>$\text{Traj}_2, \text{Traj}_3, \text{Traj}_5$</td>
</tr>
<tr>
<td>$\text{Partition}_4$</td>
<td>$\text{Traj}_5$</td>
</tr>
<tr>
<td>$\text{Partition}_5$</td>
<td>$\text{Traj}_3, \text{Traj}_5$</td>
</tr>
<tr>
<td>$\text{Partition}_6$</td>
<td>$\text{Traj}_2, \text{Traj}_3, \text{Traj}_5$</td>
</tr>
</tbody>
</table>

* $\text{Traj}_i$ represents the trajectory objects stored in database.
* $\text{Partition}_i$ denotes the partition cell identifier.
EXPERIMENTS

We have conducted a series of experiments to show the correctness and performance of the different data models and data access mechanisms. In this section, we will begin with demonstrating the datasets we have used and the experimental settings. Later, different aspects of the proposed data models and indexing strategies will be shown. The aspects include the space and time requirements of the data models as well as their performance when inserting and querying data.

Experimental Settings and Datasets

We have used three artificial datasets for testing our data models and indexing structures. The datasets are created using artificial dataset generator ERMO-DG (Aydin, Angryk, & Pillai, 2014) and include polygon-based trajectories with different spatiotemporal characteristics. For each dataset, the number of trajectories, the total number of time-geometry pairs, and the total number of points that polygon-based trajectories are comprised of are shown in Table 1.

The experiments are performed on cloud computing environment, Amazon Web Services. For all the experiments three data nodes are used for hosting tablet servers. The data node roles, as well as the other roles we have assigned nodes for our system in our experiments, are shown in Table 2. Nodes that are assigned with the role NameNode controls the distributed file system (HDFS). Zookeeper is used by Accumulo for determining the current state of processes, coordinating distributed tasks, ensuring fault tolerance, and storing configuration that can be modified on the fly without restarting Accumulo. Secondary NameNode role is an auditing system for Hadoop, performing periodical checkpoints. Accumulo master is responsible for load balancing, as well as error detection in tablet servers. The tablet server and data node roles are co-located for decreasing the network latency. Tablet servers are responsible for managing, in generic terms – reads and writes to, a subset of all tables. Data nodes simply store data in HDFS. The nodes used in AWS have all same system settings and are medium size computing instances (officially tagged as m3.xlarge) with 10-core 2.5 GHz Intel Xeon E5-2670 processors, 15 GB RAM, and 80 GB SSD storage.

Data Insertion Experiments

The first aspect of the experiments is related to the runtime performance of the proposed data models and indexing structures. The results of the experiments are presented in Figure 8. In Figure 8a, the total

<table>
<thead>
<tr>
<th>Dataset Tag</th>
<th># of Trajectories</th>
<th># of Time-Geometry Pairs</th>
<th># of Total Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>95997</td>
<td>1,200,120</td>
<td>7,336,090</td>
</tr>
<tr>
<td>B</td>
<td>209833</td>
<td>2,623,976</td>
<td>15,195,862</td>
</tr>
<tr>
<td>C</td>
<td>409421</td>
<td>5,117,811</td>
<td>33,430,017</td>
</tr>
</tbody>
</table>

Table 2. The systems settings applied in AWS for the experiments

<table>
<thead>
<tr>
<th>Node Tags</th>
<th>Assigned Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td>NameNode</td>
</tr>
<tr>
<td>Node 2</td>
<td>ZooKeeper Leader</td>
</tr>
<tr>
<td>Node 3</td>
<td>Secondary NameNode</td>
</tr>
<tr>
<td>Node 4</td>
<td>Accumulo Master</td>
</tr>
<tr>
<td>Node 5 –</td>
<td>Tablet Server &amp; DataNode</td>
</tr>
<tr>
<td>Node 6 –</td>
<td></td>
</tr>
<tr>
<td>Node 7</td>
<td></td>
</tr>
</tbody>
</table>
database insertion time (in seconds) for datasets A, B, and C are shown, for all data models along with their indexed versions. In Figure 8b, the total time spent for inserting the trajectories into G-IT index for traditional data models are demonstrated. Lastly, in Figure 8c, the total time spent for inserting the trajectories into partition inverted index for partitioned data models demonstrated.
In Figure 8a, a natural observation is that all datasets exhibit similar behaviors for each data model; namely, the size of the dataset reflects in the insertion time. For each dataset, column-fragmented traditional data model ($CT$) performed better than other data models. When compared to whole-trajectory data model ($WT$), column-fragmented data model inserts serialized polygon objects one by one, while whole-trajectory data model, serializes the entire time-geometry pairs list. We can assert that inserting bigger volumes of data takes more time. We can also see traditional data models performing better than partitioned data models during the insertion. The reason for that can be the extra time spent for preprocessing the trajectory data; namely, determining the cell partitions of trajectories or segmentation of trajectories. Additionally, the duplication strategy followed for partitioned data models is another factor that increases the amount of data to be inserted. We can also observe that the segmented-trajectory data model ($SP$) has a better insertion performance when compared to the whole-trajectory data model ($WP$). The preprocessing time spent before insertion for those two models are very similar; yet, we can assert that segmentation procedure decreased the amount of data to be inserted; as well as the time spent for the insertions. For all indexed models ($ICT, IWT, IWP, ISP$), their non-indexed versions performed better. It is much expected considering the time spent for creating, inserting, and maintaining the index structures. In Figure 8b and 8c, insertion time spent on G-IT index for traditional data models and “partition-inverted index” for partitioned data models, respectively. We can observe that both G-IT index and partition inverted index performs similarly for their respective data models. Note that, Figure 8b and 8c shows the total time spent on index insertions, and the number of trajectories in the datasets are approximately doubled every time.

**Memory and Storage Requirements Experiments**

This part of the experiments is related to memory and storage requirements of the different data models and their respective indexing structures. Our observations on storage requirements of data models in Accumulo database and the memory requirements of the indexing structures can be seen in Figure 9. In Figure 9a, database size for three database models under different data models is presented. Figure 9b and 9c demonstrate the memory requirements for traditional data models (Column-fragmented ($CT$) and Whole-trajectory ($WT$)) and partitioned data models (Whole-trajectory ($WP$), Segmented-trajectory ($SP$)), respectively. It is important to point out that the size of the database is measured using the total amount of data sent to Accumulo database during insertion phase, and the total amount of storage space being used can vary from system to system. Our database size measurements are significant due to the importance of the data volume being ingested in Accumulo. On the other hand, index sizes are measured by calculating the amount of memory space used by the internal data structures specific to Java programming language.

For all the datasets, the smallest database size is achieved using the column-fragmented traditional model for all datasets, while using whole-trajectory partitioned data model results in the largest database size. We can easily see the similarity between the insertion time requirements (shown in Figure 8a) and storage requirements (shown in Figure 9a). This is very natural, as the insertion of larger data requires more time. Our analysis for higher insertion time requirements using whole-trajectory traditional data model ($WT$) and partitioned data models ($WP, SP$) suggests that the higher volumes of data insertion (due to replications and serializations of internal data structures) causes the higher data insertion times. With our results shown in Figure 9a, it can be seen that insertion time to database and the amount of data inserted to database follows the same trend in our experiments.
In Figure 9b, the memory requirements for in-memory G-IT index can be seen. Similarly, Figure 9c demonstrates the memory requirements for the partition inverted index. For our largest dataset (Dataset C), column-fragmented traditional data model (CT) creates a little more than 2GB of raw data while whole-trajectory traditional data model (WT) creates approximately 2.9 GB of raw data. The memory
requirements for G-IT index for Dataset C (shown in Figure 9b), is approximately 3.1 KB. This result is very promising, because for over 400,000 trajectories (with only vector data that sums up to 2 to 2.9 GB), storage requirements for G-IT index are very low; therefore, it is suitable for in-memory storage. A similar situation also occurs for the partition inverted index. The amount of memory space required for partition inverted index (see Figure 9c) for Dataset C is approximately 57 MB. When compared to 4 GB database storage space needed for whole-trajectory partitioned data model and 3.1 GB space needed for the segmented-trajectory partitioned data model, it only requires 1.4% to 1.8% of actual storage in order to store the index structure.

**Spatiotemporal Query Experiments**

A major part of our experiments are related to the querying performance of our proposed data models with purely spatial, purely temporal and spatiotemporal predicates. Figure 10 demonstrates the results of our experiments. Figure 10.a and 10.b shows average query times for purely temporal queries (time-stamp and time range queries, respectively). Figure 10.c shows the average query times for purely spatial queries (for spatial window query). Lastly, Figure 10.d shows the average query times for spatiotemporal window queries. For purely temporal predicates, we experimented using timestamp and time range queries with overlap (or intersection) predicate. Purely spatial querying includes spatial window query, with a rectangle geometry query window and overlaps predicate. Spatiotemporal window queries include a rectangle spatial window and a time range. Queries are tested with traditional and partitioned data models with no indexing, as well as the indexed traditional data models. Note that we did not include indexed partitioned data models as partition inverted index is designed for non-spatiotemporal queries.

As expected, for all types of queries included in our experiments, partitioned data models worked a lot more efficient than the traditional data models without indexing structures. The reason for that is that it requires a full scan to perform a spatiotemporal (or purely temporal and spatial) query for traditional data models. On the other hand, as we encode the spatiotemporal characteristics to row identifier (which can be searched efficiently in Accumulo) in the partitioned data models, partitioned data models only require looking up very few records stored in the database. The purely spatial query type requires more time for traditional data models without indexing because more trajectories are returned. Note that, temporal dimension is larger than spatial dimensions.

Another important observation for all types of queries in these experiments is indexed traditional data models provide very similar querying characteristics with partitioned data models. For many of the queries, indexed traditional data models (especially the column-fragmented traditional model (ICT)), perform better. We can also see that query times scale well in the means of time requirements. With different sized datasets (See Table 1), there is not much difference in the average query time.

**Non-Spatiotemporal Query Experiments**

Last part of the experiments is related to the performance of the partition inverted index for queries with no spatiotemporal characteristics. In this part, we have experimented whole-trajectory partitioned data model with the partition-inverted index. For the partitioned inverted index, a multi-map data structure is used. Figure 11 shows the results of our experiments. First two sets of columns show the average query time spent on performing an exact query. The last column shows total time spent on inserting data to database using indexed whole-trajectory partitioned data model.
It is fair to state that partitioned data are not much appropriate models for non-spatiotemporal queries in the means of runtime performance. It can be seen that even the insertion time for indexed partitioned data models is much less than the average time spent on searching for a single record. Similar to traditional models' behavior on spatiotemporal queries, partitioned data models provide a poor schema for non-spatiotemporal queries. We can also argue that traditional data models answer the spatiotemporal queries faster than partitioned data models answer the non-spatiotemporal queries, because of the
filter-and-refine strategy applied in the search algorithms of traditional data models (See Figure 10). Nevertheless, partitioned data models, using the partition-inverted index, can be tuned for better non-spatiotemporal query performance.

CONCLUSION

In this work, we addressed the problem of storing spatiotemporal trajectories in non-relational databases. For this problem, we have proposed two types of data models: (1) traditional data models, and (2) partitioned data models. Traditional data models mimic the classic relational database models by using the trajectory identifier as the row identifier in the database. On the other hand, partitioned data models use a partitioning algorithm, in order to find the spatial and temporal location of the trajectories; and, use it as the row identifier.

In addition to the data models, we have designed two in-memory indexing structures for providing efficient querying capabilities. For traditional data models, we have designed the G-IT spatiotemporal index, to strengthen the weak spatiotemporal querying performance of these models. For partitioned data models, we have designed the partition inverted index, which can be used for efficiently performing non-spatiotemporal queries.

Our experiments show that in the means of storage requirements, traditional data models are better. On the other hand, for spatiotemporal queries, partitioned data models are preferable if no index is to be used; yet, for non-spatiotemporal queries, partitioned data models perform poorly. Nevertheless, traditional data models with the G-IT indexing perform very similarly to partitioned data models. In a similar way, partitioned data models’ non-spatiotemporal query performance can be increased using the partition-inverted index. In No SQL databases, data modeling is primarily driven by the nature of accessing the data and that is the reason they are called question-focused datasets.
For future work, our aim is to provide efficient join algorithms, especially for spatiotemporal predicates. Many relational database management system vendors provide this capability; however, non-relational databases provide better distributed computing model. In the era of big data, it is important to use non-relational databases for mining algorithms. The spatiotemporal join procedures are vital for many spatial/spatiotemporal data mining algorithms, as the performance of many spatiotemporal pattern-mining algorithms primarily rely on the efficient spatial and spatiotemporal join mechanisms.

REFERENCES


Chakka, V. P., Everspaugh, A., & Patel, J. M. (2003). Indexing Large Trajectory Data Sets With SETI. CIDR.


Modeling and Indexing Spatiotemporal Trajectory Data in Non-Relational Databases


**KEY TERM AND DEFINITIONS**

**Interval Tree**: A tree-based ordered data structure used for the efficient retrieval of time intervals.

**Inverted Index**: A map-based indexing structure originally used for indexing words in a document. The structure provides a mapping from content to its location in the database.

**Minimum Bounding Rectangle**: Maximum extents of a 2-dimensional spatial object within a specific coordinate system.

**Non-Relational Database**: A class of database systems that does not follow the constraints and the rules of relational model, for providing simple storage and retrieval mechanisms for mostly large scale and real time web applications.

**Partitioned Data Model**: A class of data models for storing spatiotemporal trajectories in non-relational database systems that uses the spatiotemporal characteristics of trajectories by initially applying a fixed partitioning.

**Time-Geometry Pair**: An ordered pair of timestamp and geometry, representing a snapshot (geometric representation) of trajectory at a particular timestamp.
**Traditional Data Model:** A class of data models for storing spatiotemporal trajectories in non-relational databases systems that mainly uses non-spatiotemporal characteristics of trajectories.

**Well-Known Binary:** A portable representation of spatial geometries that are formed by contiguous stream of bytes.