Spatio-temporal Indexing in Non-relational Distributed Databases

Anthony Fox, Chris Eichelberger, James Hughes, Skylar Lyon

Abstract—Big Data has driven the need for datastores that can scale horizontally leading to the development of many different NoSQL database implementations, each with different persistence and query philosophies. Spatio-temporal data such as location data is one of the largest types of data being collected today. We describe a novel spatio-temporal index structure that leverages the horizontal scalability of NoSQL databases to achieve performant query and transformation semantics. We present performance characteristics gathered from testing with Accumulo.

Keywords—spatio-temporal, nosql, big data, column family database, geohash

I. INTRODUCTION

Spatio-temporal data sets have seen a rapid expansion in volume and velocity due to recent web services capturing geolocations for much of user activity. Tweets on Twitter have a spatio-temporal reference as do photographs uploaded to Instagram and Flickr. Foursquare allows users to ‘check-in’ to a location at a given time. Most of these data sets have become possible with the advent of smartphones that double as geolocating sensors. Traditional RDBMSs can no longer keep up with the volume of data, and thus researchers and industry have begun exploring alternative persistence and query technologies.

Our goal in this paper is to present a spatio-temporal indexing structure built on top of a column-family oriented distributed data store that enables efficient storage, querying, and transformation capabilities for large spatio-temporal data sets. First, in section 2, we will review spatial indexing strategies, geohashes, and non-relational data stores with a focus on column-family stores. In section 3, we discuss our strategy for storing and retrieving geo-time data in Accumulo, an instance of a column family data store. In the last two sections, we report the test results, draw conclusions, and indicate directions for future effort.

II. BACKGROUND

Before we discuss higher dimensional indexing, we recall that $B^+$-trees are used widely in file systems and databases to store and retrieve files and entries effectively. $B^+$-trees provide a tree structure on linearly ordered data such as a time field in a database table. While $B^+$-trees and various derivative data structures cannot directly store higher dimensional data, they still play an integral role organizing the data structures which can serve to index multi-dimensional entries in a database. Of note, both the $R$-trees we discuss next and Accumulo use $B^+$-trees to store data. After a discussion of indexing spatial data, we will recall the details of geohashes which form the basis of the crucial spatial component of our Accumulo key design. Lastly, we will give some background information about Accumulo.

A. Spatial indexing in RDBMSs

A spatial database is a database specially equipped to store data with a geometric component and to retrieve results using topological and distance-based queries. Typical examples of queries include topological predicates such as “covers” (e.g., “find police stations in Chicago”) or “intersects” (e.g., “find rivers which run through Tennessee”) as well as metric-based queries like finding all entries within a distance of a point (a range query) or finding the $k$ nearest neighbors to a geometry.

In traditional RDBMSs, the entries are stored in a table, and an additional spatial index is built separately. This index can be referenced by the database system and provides the chief means of efficiently answering queries which contain a geometric predicate.

Many traditional RDBMSs employ $R$-trees or QuadTrees for indexing, so we will recall their basic details next. In particular, PostGIS adds $R$-tree support to PostgreSQL[17].

1) $R$-trees and QuadTrees: In general, an $R$-tree stores n-dimensional geometries by replacing each geometry with its minimum bounding (n-dimensional) rectangle (MBR). The MBRs are stored in a $B^+$-tree structure. Since Guttman’s original paper describing $R$-trees[11], numerous modifications have been suggested and implemented[15]. These $R$-tree variants improve storage and hence retrieval time in exchange for complexity in inserting, deleting, and maintaining the $R$-tree. Separate from the particulars of tree management, algorithms have been designed to address specific requests including range queries[11], topological queries[15], and $k$ nearest neighbor[18]. (Again, see [15] for a survey.)

Finkel and Bentley defined quadrees in [8]. A quadree is a tree where each non-leaf node has exactly four children. This structure allows one to split a rectangular region into quarters in a natural way. Oracle Spatial is an example of a RDBMS with Quadtree-based-index support. In [13], researchers at Oracle found that their $R$-tree index outperformed their Quadtree index for their test sets. Further, they noted that it required extensive testing to find the optimal tiling level for optimal Quadtree performance.

2) Other Recent Approaches: In recent years, document oriented databases have seen increased use in scale-challenged scenarios. Two such databases of particular interest to the spatial community are MongoDB and Solr. Both have incorporated...
rated geohashes in their spatial indexing approaches[1], [19], so we recall the details of geohashes next.

B. Geohashes

Gustavo Niemeyer invented geohashes in 2008 with the purpose of geocoding specific points as a short string to be used in web URLs (http://www.geohash.org/). He entered the system into the public domain by publishing a Wikipedia page on February 26, 2008[10].

A geohash is a binary string in which each character indicates alternating divisions of the global longitude-latitude rectangle $[-180, 180] \times [-90, 90]$. The first division splits the rectangle into two squares ($[-180, 0] \times [-90, 90]$ and $[0, 180] \times [-90, 90]$). Points (or more generally geometries) which are to the left of the vertical division have a geohash beginning with a ‘0’ and the ones in the right half have geohashes beginning with a ‘1’. In each of the squares, the next split is horizontal; points below the line receive a ‘0’ and the ones above a ‘1’. This splitting continues until the desired resolution is achieved.

In order to make geohashes more useful for the web, the inventor assigned a plain text, base-32 encoding for his web service. As binary strings, geohashes can be of any non-negative length, but for web use, geohashes are typically seen in lengths which are multiples of five.

Geohashes provide some notable properties.

- Each geohash can be thought of as a longitude-latitude rectangle.
- Geohashes provide a $z$-order traversal of rectangles covering the Earth at each resolution. (Figure 1)
- Containment. Adding characters to the end of a geohash specifies a smaller rectangle contained inside the initial one. $t \supset tt \supset ttuv$. (Figure 2)
- Locality. Shared prefixes imply closeness. ($dp$ is close to $dr$). Note that the converse is false as $9z$ and $dp$ are adjacent while having no common prefix. (Figure 2)

Geohashes effectively define an implicit, recursive quadtree over the world-wide longitude-latitude rectangle. We will leverage this trie structure to construct spatial keys in Accumulo.

C. Accumulo: A Non-relational Distributed Database

Accumulo was inspired by Google’s BigTable implementation. We will briefly present some BigTable features, and then describe a few notable additions in Accumulo.

1) BigTable Database Model: BigTable stores its data as a sorted list of key-value pairs where the keys consist of a row key, a column key, and a timestamp; and values are byte arrays. Each table consists of a number of tablets, each of which holds the data for a lexicographical range of row keys. These tablets are stored in column-oriented files in the Google File System, Google’s proprietary distributed file system. To further assist with locality, column keys consist of a column family and a column qualifier. For each table, a moderate number of column families are specified and those entries are stored together. Thus column families allow a BigTable user to reason about how their data is stored. The last part of the key, the timestamp can be used for version control.

Since the publication of the BigTable design by Google[5], a number of other column family oriented databases have been developed and released. Three of the most popular are Accumulo, Cassandra, and HBase; each is a top-level Apache project. While each has strengths and weaknesses, we will focus on Accumulo because of its ability to add server-side programming via a feature called iterators.

2) Accumulo Overview: Apache Accumulo builds on the BigTable design by adding cell-level security and a server-side programming model. Our spatial query capability leverages iterators to execute spatio-temporal predicate queries. Figure 4 shows a representation of an Accumulo key-value pair. Note that Accumulo keys have added a visibility component to BigTable’s keys and explicitly split the column key into the column family and column qualifier. The column visibility entry consists of optional authorization tokens; only clients with the proper combination of authorizations are allowed to
read that cell, providing for granular access controls at the cell level.

![Fig. 3. Structure of a key-value pair](image)

3) Accumulo Iterators: The Accumulo core/system iterators provide a sorted view of the key-value pairs in the table. User-defined/user-configured iterators provide two main functions: filtering data and transforming data. As an example, we could apply a timestamp filter to return only the entries in a given time period. Additionally, we could add a Combiner to produce statistics (such as the maximum value, the sum, and the count) on the filtered rows. When iterators are composed, we have an iterator chain.

Accumulo enables the use of a pattern known as the Intersecting Iterator to colocate and traverse an index and the associated data[9]. We depend on this pattern in our spatial index, and we briefly describe it here in the context of an inverted index for text searching of a document. There are two related key formats in this pattern. The index format contains a bin number in the row. This effectively acts as a shard index and ensures that all of the information associated with that bin are stored on a single tablet on a single server, because Accumulo never splits rows across tablets. The column family contains a single term while the column qualifier contains a document ID.

![Fig. 4. Index Row Key Format](image)

The actual document is stored with the same bin number so that it resides in the same row (and hence tablet) as the index entry. Each document row’s column family contains the document id and corresponds to the column qualifier of the index cell.

![Fig. 5. Data Row Key Format](image)

As an example, to perform a query such as “Select all documents that contain the word ‘dutiful’”, a client uses two coordinated server-side iterators. The first iterator traverses the index keys and emits the column qualifiers for hits. The second iterator traverses the data keys and returns key-value pairs with column families returned by the first iterator.

4) Bloom Filters: Originally proposed by Burton Bloom [2], a Bloom filter is a lossy mechanism for determining whether a piece of information has ever been encountered. The filter consists of a collection of binary probes that are initialized to 0, but will remain set to 1 once they are activated. When a new object arrives, multiple hash functions are used to associate that object to a subset of the probes, and those probes are activated. To test whether a new object has ever been seen before, its probe set is identified and checked: If not all of the probes in the probe-set are activated, then the object has never before been seen by the Bloom filter. A negative response indicates definitively that the object has never been processed by the filter; a positive response indicates only that it is possible (but not certain) that the object was previously encountered. The likelihood of a Bloom filter reporting a false-positive increases directly with the number of objects stored, and decreases with the number of probes allocated. A Bloom filter is maintained for each block in a tablet and is used to filter out requests which will not match any entries in the block.

5) Accumulo Load Balancing: Load balancing is the process of allocating activity and resources across workers so that no single worker is significantly busier than any other for any long span of time. Within a key-value store, activity follows data, so load balancing becomes a matter of how to distribute data.

Accumulo’s TableLoadBalancer works by spreading each table’s tablets across the tablet-servers as evenly as possible. This is particularly useful when the underlying data are randomly sharded (as is the case with our software), because the random sharding shares the same goal as the table-balancing: Distribute query work across all of the available nodes as evenly as possible. Because its advantages favor our use case, we use the TableLoadBalancer rather than the default load balancer.

III. Spatio-Temporal Index Structure

There is no perfect way to map from three dimensions (latitude, longitude, time) to one (lexicographical ordering of keys in a table), especially when time has radically different bounds than location. Naively, one might think to store each geo-time object three times: once ordered by latitude; once ordered by longitude; once ordered by time. The approach does not scale well, nor should it: The power of a key-value store is in its distribution, so for a query scheme to work well, it must be able to divide the work among the participating nodes. Computing the intersection among three independently-ordered sets requires either moving data between nodes (slow); chaining searches (slow); or centralizing sub-query results for a coordinated join (slow and memory-consuming).

A relational database (RDBMS) has the ability to use information from multiple indexes to determine how best to search for the records that satisfy all query criteria. A
NoSQL key-value store, in contrast, has only a single index that is built atop the constraint that all records are ordered lexicographically. This means that the accumulo-geo indexing scheme is essentially a way to encode geo-time information in rows so that their natural ordering innately makes any query – consisting of a search polygon and a date/time interval – quickly reducible to a set of range requests that contain the desired results and a minimum of additional (non-qualifying) elements.

We build index keys by interleaving portions of a points’ geohash string representing the geometry with parts of the datet ime string, and prefix the row identifier with a random bin number to help spread our data across the tablet servers. By choosing to use 35-bit geohash strings and ‘yyyyMMdd’ representations of dates, this construction divides the data into unit compartments that are approximately 150 meters square and one hour in duration. Geohashes alone proceed through space in a z-order traversal, and date strings proceed linearly through time. The way that parts of these two elements are woven together yields a linear order that stutter-steps its way through every compartment in the three-dimensional space.

A. Storing Point Data

Let us consider storing data associated to a point and a time. See figure 6. The location of the point is represented as a 35-bit geohash (a 7-character string using Niemeyer’s encoding), and the date-time information is encoded as a string of the format ‘yyyyMMdd’. These two strings – one for location and one for time – are divided up into parts, and distributed among the elements of the index key in a manner described shortly.

If we only used geohashes and dates to index our entries, some rows of our tables would contain vastly more data than others. If one were storing geo-located tweets, for example, New York City would have many more entries than Charlottesville, Virginia. Because each row is stored in exactly one tablet (and hence, one server), concentrating a large number of similar entries in a single row would subject that server to a disproportionate query load, bogging down response times.

1) Index Key Format:

- **Row Key**
  To avoid such an imbalance, our index keys begin with a random bin number in a designated range to act as a sharding parameter. Appended to the bin number in the row key, we add the coarsest geospatial and temporal bound, separated by a specially designated character. This distributes similar geo-time points uniformly across the shards, and enables pre-defined split-points for the table – based on the maximum shard number – ensuring that all queries are parallelizable. For example, 01 u 201205 is a row key that specifies a shard of 01, a geohash bound corresponding to the u rectangle, and the month of May 2012.

- **Column Family and Qualifier Key**
  The column family of the index key contains the next resolution of geospatial bounds, while the column qualifier contains the identifier of the data element and even higher-resolution spatio-temporal bounds. For example, if the column family is 01m and the column qualifier is GDELT.2973011______ tw0 0722, then, in conjunction with the row key, the data element with id GDELT.2973011 (padded with underscores) falls within the u01mtw0 35-bit geohash and during the 10:00pm hour of May 7, 2012 (UTC).

2) Data Keys: Every index-key, in accordance with the intersecting-iterator pattern, has one corresponding data key. The data key shares the Row ID with the index key, ensuring colocation between the index-keys and data-keys, but uses the element identifier as its column family; uses an attribute name as its column qualifier; and stores the fully-encoded form of the SimpleFeature representation of the object in the value.

B. Storing non-point geometries

To store more complex geometries such as linestrings and polygons, the storage engine decomposes each non-point geometry into a collection of covering, disjoint geohashes at potentially mixed resolutions, and stores each constituent geohash as a separate item associated with the common identifier. Figure 7 illustrates how a polygon and line-string are decomposed into subordinate geohashes.

[Fig. 6. Encoding geo-time data into an index entry]

[Fig. 7. The decomposition algorithm balances using a small number of multi-resolution geohashes which cover the geometry with reducing the amount of “wasted” overlap.]

At query time, the common identifier is used to filter out duplicate entries so that the user is presented with exactly one copy of each unique geo-time object.

1This geo-time API relies upon many of the Open Geographic Consortium standards as embodied in the GeoTools library. SimpleFeature is a GeoTools abstraction of a geographic object with support for other attribute, value pairs.
C. Iterator Chain/Query Planning

Our iterator stack consists of three layers (Figure 8) designed to remove entries which do not match our queries as soon as possible:

1) Accumulo’s internal iterators. The query planners use key-range and column family iterators to perform coarse geo-time filtering.
2) Geo-time iterators and aggregators. The geo-time filtering iterator decodes each index key into its original geohash and datetime string. If the key is within the query, the entry is sent to the aggregator which recombines the corresponding data rows into a SimpleFeature.
3) Feature filter. If there is an attribute filter (expressed in ECQL), the feature filter applies it to the SimpleFeature from the previous step.

One significant aspect of the Accumulo iterators is that they are executed in parallel across the tablet servers, so this entire stack is acting independently on every machine in the cluster. As a consequence, there is very little central coordination to impede the flow of results to the query client.

One significant aspect of the Accumulo iterators is that they are executed in parallel across the tablet servers, so this entire stack is acting independently on every machine in the cluster. As a consequence, there is very little central coordination to impede the flow of results to the query client.

![Figure 8. The iterator stack](image)

Fig. 8. The iterator stack

1) Query planning: Query planning is, beyond the trivial cases in which there is no query polygon or interval, largely a matter of translating query parameters into iterator settings. There are two prongs to this translation: Row-ranges and column-family values.

Row-ranges can be built up from the constituent parts. The advantage of doing so is that Accumulo manages index blocks which record where various row values begin throughout the table, facilitating random access. Remember that the keys are constructed (simplistically) as follows:

- Row ID: random shard number; the first character of the 7-character geohash; year and month of date/time
- Column Family: characters 2, 3 and 4 of the geohash string
- Column Qualifier: the entry ID; characters 5, 6, and 7 of the geohash; day and hour of date/time

Each of these elements has a discrete number of possible values, though for some that number is quite high. Consider a date range of January 1, 1990 through December 31, 2010, in which the total number of distinct hours is too large to be useful. In these cases, a range is substituted for an enumeration of values. The planner assembles the collection of enumerations and ranges for all of the key parts, and then determines how best to coalesce these into a single list of key-ranges that – when taken together – will necessarily contain all of the entries that satisfy this query.

The most important consideration for this consolidation is the number of entries: If the query polygon fit inside a single 35-bit geohash, and the query interval fit inside a single hour, then there need be only as many row ranges as there are shards. For other queries, the planner often has to compromise by selecting wider ranges in order to accommodate the combinations of key elements that might satisfy a query. In these cases, the final row-ranges selected are more general, meaning that the percentage of query misses that fall inside may be higher than in more focused ranges.

Separate from the row-ranges, the column family can serve as the basis for an independent portion of the query plan. There are two reasons why this is useful:

1) Accumulo has built in facilities for specifying which column-family values should be accepted by the default iterators;
2) a Bloom filter – established when the managed table is created – is configured on the Column Family.

The column family consists of three geohash characters, so the number of different column families values is 32,768. In practice, there are typically far fewer combinations that can satisfy any given query, since most query-polygons are smaller than a 5-bit geohash. Most of the query responses will share at least the first two characters of their geohashes, reducing the number of possible column-family values to less than or equal to 1,024.

In addition to directly selecting the column family values, though, there is a secondary benefit: the Bloom filter applied to the column family. When the block-index is consulted by Accumulo during a seek, it can inquire of the Bloom filter – in advance of reading any values from that block – whether it is possible that any of the desired column-family values exist inside. If the Bloom filter indicates that none of the desired column-family values have been added to the block, then that block can be skipped entirely. (If the Bloom filter indicates that one or more of the values may have been used, then a full block-scan must be performed, even though the Bloom filter may have been wrong. See section Bloom Filters).

D. Implementation Details

To facilitate use of this spatial indexing system, we have implemented the Geotools DataStore, FeatureStore, and FeatureSource interfaces. This has allowed us to configure GeoServer, an open-source WMS service (and much more), to generate rasterized tiles of high volume spatio-temporal data backed by Accumulo. In effect, this opens the spatio-temporal data set to access by any OGC-compliant client and, coupled with the OGC ECQL standard, provides a flexible and standards-based means of interacting with these data sets.

If the query polygon spans 5-bit geohashes, even if small, then the number of column-family values will likely explode to its full 15 bits.
IV. Performance Results

A. Test data

For our testing, we chose the [Global Database of Events, Language, and Tone][1] (GDELT). This dataset consists of over 200 million geo-political events from across the globe coded using CAMEO codes describing the type of event. We picked this dataset since it represented a range of temporal and spatial data. Additionally, since the data represent automatically extracted international events from around the world, some places would be represented more often giving the dataset a representative non-uniform spatial distribution. In Figure 9 and 10, we can see this distribution across a map of the world and by geohash.

Fig. 9. GDELT event counts aggregated per 21-bit geohash (colored by quantile)

Fig. 10. Distribution of GDELT events across 21-bit geohashes

B. Queries

There were four broad categories of experiment conducted as part of this research: spatial queries; temporal queries; filtering queries; and scaling.

1) Spatial: This experiment was designed to capture the relationship between the number of points in a polygon and the query’s total response time. We collected a group of 105 polygons from around the world; see Figure 11 for their geographic distribution. Each of these polygons is corresponds to a single geohash, most specified at 35 bits. A few larger geohash polygons were included for completeness. Furthermore, each of these 105 polygons was specifically chosen because it contains at least 100 data points.

The polygons were presented 10 times each, and their order was randomized before each replication. Each polygon became the basis for a single geo-time query, using a static interval of calendar year 2011. Queries were presented sequentially, so as to reduce side-effects, keep timing consistent, and normalize over cached results. The results of this experiment are presented in Figure 12.

Fig. 11. Distribution of our standard test-set of polygons sampled from the larger GDELT database. Most of these polygons represent specific 35-bit geohashes, though there are a few larger polygons included in the set.

Fig. 12. Response times for a standard set of global query-polygons. This is a log-log plot to highlight the lower end of the scales.

The upward trend simply reflects the fact that increasing the number of query responses increases the response time, and would be notable only if it were not present. Somewhat more interesting is how the variation in the timing results appears to be greater among the low-volume queries than it is among the high-volume queries. This is almost entirely an artifact of the way the plot axes are scaled: Because there are multiple orders of magnitude in both query density and response time, and because the lower-end of both axes is significantly more crowded than the upper-end, the data are rendered on a log-log plot to allow for better visual separation. This has the side effect of amplifying the visual impact of what are genuinely minor changes in the mean response time.

The main result from this experiment is that the indexing scheme was able to complete most of these queries in one second or less. Another consequence of Accumulo’s batch-scanner implementation is that results stream back to the client as soon as they become available. Therefore, initial results arrive sooner than the time reported for the query to complete.

2) Temporal: These experiments were designed to measure the impact of temporal filters of different sizes. The location – a rectangle representing a coarse bounding box around the city of London, England – was held constant. Three separate window-sizes were selected: one day; one week; and...
one month. Twenty-one periods of each size were evenly
distributed across an interval from 2012-06-01 to 2012-12-
31, and each unique query interval was run one time. The
results are summarized in Table I that includes the number of
responses and the effective throughput defined in terms of the
number of records returned per second.

### Table I
Temporal Results

<table>
<thead>
<tr>
<th>period size</th>
<th>mean responses</th>
<th>mean throughput (records/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
<td>436</td>
<td>2,019</td>
</tr>
<tr>
<td>week</td>
<td>1646</td>
<td>8,034</td>
</tr>
<tr>
<td>month</td>
<td>4564</td>
<td>32,988</td>
</tr>
</tbody>
</table>

3) Filtering: There are three generic types of queries that
can be applied to geo-time data:

1) **Spatial**: constrain the location with a polygon: “select
only the items in Virginia”;
2) **Temporal**: constrain the time with a date-time interval:
   “select only the items in December 2012”;
3) **Attribute Filtering**: constrain the results based on non-
geo-time attributes specific to the data set: “select only
the items having a price less than $100 or a product type
other than ‘LUXURY’”

The aim of the filtering tests was to examine the relative
costs of these three types of constraints. Four separate sets of
test were conducted:

#### Table II
Attribute Filters

<table>
<thead>
<tr>
<th>polygon</th>
<th>interval</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>London</td>
<td>August 2012</td>
<td>EventBaseCode='010'</td>
</tr>
<tr>
<td>London</td>
<td>August 2012</td>
<td>EventBaseCode='010'</td>
</tr>
</tbody>
</table>

The attribute filter is written using the Extended Common
Query Language (ECQL) [7], part of the API offered by
GeoTools. Though it can specify queries using spatial and
temporal clauses, the use-case presented here relies solely
upon the non-geo-spatial filtering capabilities of the language.
The GDELT CAMEO event codes are divided into a coarse
root code, a more specific base code, and finally, the full event
code. We used the attribute-filter =EventBaseCode='010'=, cor-
responding to “Make statement, not specified below”. Such
events are generic, and occur fairly frequently (about 10% of
entries), making it a good candidate for testing.

4) Scaling: The purpose of this experiment was to investi-
gate how geo-time queries scale with the number of available
tablet-servers, and to identify what other factors influence
response time. Our base infrastructure consists of 13 virtual
machines, each hosting its own tablet server. A set of 1,000
separate polygons constituted our query set. The independent
variables included:

1) the number of tablet servers running: 13 (all) or 6
2) the number of shards: 100 (default) or 26
3) the number of geohash characters to store in the Row
   ID (spatial bin level): 1 (default) or 2

![Fig. 13. Attribute-filtering queries](image)

Table III summarizes the results from running the query
set against all 8 of these configurations. The data in the
corresponding figure have been transformed, in that the X-
axis reports the mean number of tablets per tablet-server, and
the Y-axis reports the mean throughput of the configuration
in records per second. This change-of-variables makes the
underlying trends in the data more readily apparent.

#### Table III
Comparison of Number of Tablet Servers, Number of Shards,
and Number of Geohash Characters in the Row ID.

<table>
<thead>
<tr>
<th>tablet servers</th>
<th>parts</th>
<th>spatial bins</th>
<th>mean time (sec)</th>
<th>mean records/sec</th>
<th>tablets /server</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>26</td>
<td>1</td>
<td>1.82</td>
<td>2,840</td>
<td>2.0</td>
</tr>
<tr>
<td>13</td>
<td>26</td>
<td>2</td>
<td>0.93</td>
<td>16,907</td>
<td>2.0</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td>1</td>
<td>1.92</td>
<td>2,518</td>
<td>4.3</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
<td>2</td>
<td>1.26</td>
<td>12,709</td>
<td>4.3</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>1</td>
<td>1.88</td>
<td>2,466</td>
<td>7.7</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>2</td>
<td>0.99</td>
<td>11,117</td>
<td>7.7</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>1</td>
<td>2.24</td>
<td>2,016</td>
<td>16.7</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>2</td>
<td>1.12</td>
<td>8,755</td>
<td>16.7</td>
</tr>
</tbody>
</table>

The number of running tablet servers appears to have a
weak (< 10%) affect on query-response time. Given that most
of the query polygons are relatively small and have few hits
each, the time spent planning each query may well dominate
the time required to execute the query. Having more running
tablet servers improves query performance, but not without
friction.

The number of shards was not important independently, but
the relationship to the number of running tablet servers appears
to have a small effect on query throughput, with lower ratios
being weakly associated with greater throughput. This trend
is consistent across both 1- and 2-geohash-characters in the
Row ID. Considering that the query planner creates at least
one range – and often more – per shard and that all of the
tables necessarily must be scanned as part of every query, it
seems reasonable that loading more tablets on to each tablet-
server would degrade the performance of the queries.

The most significant variable appears to be the number of
geohash characters that are incorporated into the Row ID,
with 2 consistently out-performing 1. One geohash character
V. RESULTS ANALYSIS/CONCLUSION

In traditional RDBMSs, $R$-trees (and other spatial indices) are external to the data and must be updated and computed when new features are added. This is computationally intensive and affects performance adversely as data size increases. Our geohash-based approach allows us to calculate a key for each piece of data in advance. Under this method, inserting and deleting an entry uses the regular methods for adding and removing data from Accumulo, so we incur no additional overhead. The distribution of data is built into the index schema by varying the number of bins and the coarsest spatio-temporal resolution.

As expected of the temporal queries, the largest window – a month – takes the longest to complete. Of greater interest is the fact that there is relatively little separation in the query-times for the day and week periods. This is, in part, attributable to the indexing scheme. Each row is identified by the event year and month; units of time smaller than the month are not encoded until the column qualifier. This means that row-ranges are likely to use the time down to the month, but not necessarily much finer-grained than that.

That there is so little difference in the response-times between the day-long intervals and the week-long intervals suggests that either:

1) both intervals used the same query plan (meaning that they shared the same candidate set), and the time required to filter this candidate set is more significant than the time to return query hits; or

2) the intervals used different query plans, but those plans implicated the same data blocks, in which case there was only an insignificant difference in the work to be done.

From our testing, we observed that query predicates did result in queries which returned faster since fewer results were returned. These additional filters created additional work for the back-end, reducing throughput, and Accumulo could leverage the distributed disk and processing to realize a substantial speed-up in the rate of return. We observed that temporal filtering is less demanding than attribute-filtering. This is because the temporal information is incorporated in the index rows, while the attribute information is stored within the encoded Feature as the value of the data rows.

Tuning the number of bins and the resolution levels has an impact on the performance of the spatio-temporal index. The number of bins should roughly correspond to the number of tablet servers so as to ensure parallelized query result computations. However, the expected growth of the table should be taken into account when defining the number of bins. Since a crucial advantage of distributed data stores over traditional RDBMs is horizontal scalability, the store can grow into the number of bins by adding tablet servers at runtime. Furthermore, if the geospatial domain of data is a subset of the whole world, then the coarsest spatial resolution should be tuned accordingly. The same applies to the temporal dimension - if data falls across decades, then year (and possibly month) should go into the row key, while if data falls within a year, then the addition of day in the row key will improve performance of typical queries.

A. Future Work

In this paper, we have performed a basic benchmark of our spatio-temporal software based on Accumulo. We have not had a chance to experiment as fully as we would have liked with Accumulo, and we would like to outline our future directions. First, we would like to observe that while tuning traditional RDBMSs is well understood, a body of standard advice...
for tuning Accumulo has not been created and commonly accepted. In terms of scalability, we currently have a small cloud shared for development. We would like to experiment on a larger cloud to show how our software scales.

In terms of our design, there are a number of parameters which will help us plan to explore in the future. First, in terms of key and query design, given results from the team Oracle with quadtrees \[13\], we believe that query performance may vary when the geohash resolution of decomposed query polygons and the geohash resolution used to store data are different. So far, we have had acceptable results storing our data with at the 35-bit resolution, but we haven’t directly studied how different queries decompose and their effect on query times. Further in section \textbf{Spatio-Temporal Index Structure} we discussed our use of random partitions to spread out data across the tablet servers to increase participation in returning queries. During our initial testing, when we adopted the binning strategy with 100 bins, we noticed a sizable performance increase, but we do not yet have conclusive results regarding a recommended ratio of bins to tablet servers.

In \textbf{Geohashes} we reason that our key design maps geo-time into a linear z-order by interleaving temporal sub-bands and geohashes. Based on the resolution of a query’s time component, different key designs may perform differently. This is close to the idea of comparing spatial queries to the storage geohash resolution. If similar queries can be predicted ahead of time, these studies would help inform key design. Since potential database use might be unknown, we are also actively pursuing ideas which would help use create functionality akin to Postgres’s \textit{“VACUUM ANALYZE”}.

We have used the tools that were developed as part of the indexing scheme to implement a nearest-neighbor operator. There already exist good \textit{R}-tree algorithms for returning \(k\)-nearest-neighbors efficiently \[14\], but the challenge is to develop a good approach for a distributed key-value store. Our current operator accepts a query point and a bounding (polygon, interval) pair, and will return zero or more points that satisfy those constraints by aggregating the results server side. Each tablet computes its nearest-neighbor to the query point and returns that to the client. The client finds the actual nearest neighbor by computing the minimum of the relatively few results. There are a number of refinements to this nearest-neighbor approach currently being investigated, but the existing implementation serves as proof-of-concept that the library is adequate to support a rich variety of geo-time extensions.

Spatial transformations such as projection/reprojection, distance queries, and translations can all be implemented on top of the iterator chain by stacking additional operations that perform the desired computation.

\section*{References}
\begin{thebibliography}{19}
\bibitem[1]{2d} 2d Indexes - MongoDB Manual 2.4.4. \url{http://docs.mongodb.org/manual/core/2d/} [Online; accessed 20-June-2013].
\bibitem[3]{4} CAMEO Event Data Codebook. \url{http://eventdata.psu.edu/data.dir/cameo.html} [Online; accessed 20-June-2013].
\end{thebibliography}