Spatio-temporal Indexing in Non-relational Distributed Databases

Anthony Fox, Chris Eichelberger, James Hughes, Skylar Lyon
Commonwealth Computer Research, Inc
{anthonyfox, chriseichelberger, jameshughes, skylarlyon}@ccri.com

***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 [6] 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 [16], and k nearest neighbor [18]. (Again, see [15] for a survey.)

Finkel and Bentley defined quadtrees in [8]. A quadtree 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 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 **??**)
- Containment. Adding characters to the end of a geohash specifies a smaller rectangle contained inside the initial one. $\mathbf{t} \supset \mathbf{tt} \supset \mathbf{ttuv}$. (Figure 2)
- Locality. Shared prefixes imply closeness. ($\mathbf{dp}$ is close to $\mathbf{dr}$). Note that the converse is false as $\mathbf{9z}$ and $\mathbf{dp}$ are adjacent while having no common prefix. (Figure 2)

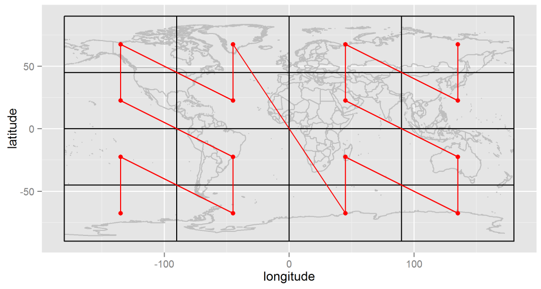

Fig. 1. z-order traversal of the globe via 4-bit geohashes. Every geohash rectangle can be decomposed into four sub-hashes that are visited in a self-similar Z pattern, ad infinitum.

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.

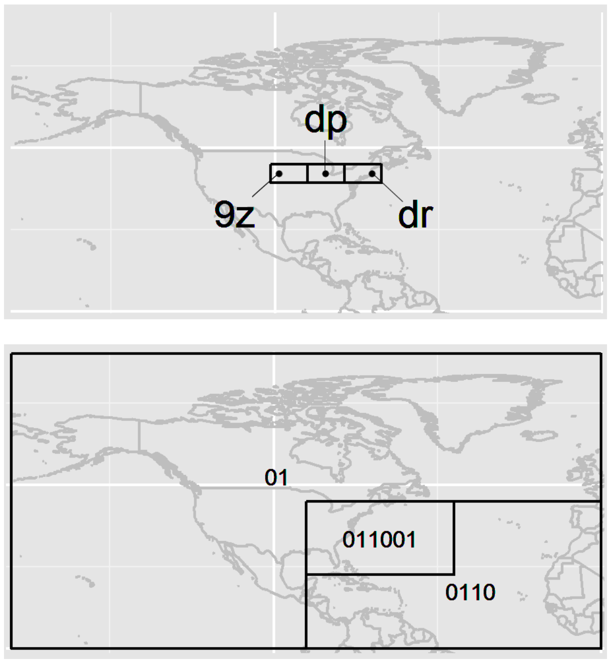

Fig. 2. Geohash Locality and Nesting

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 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 3 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. An Accumulo table forms a sparse, sorted, multi-dimensional map.

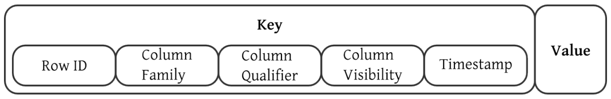

Fig. 3. Structure of a key-value pair

3) Accumulo Iterators: The Accumulo core/system iterators provide a sorted view of the key-value pairs in the table and enable seeks across the row or column family dimensions. 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 utilize a modification of 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.

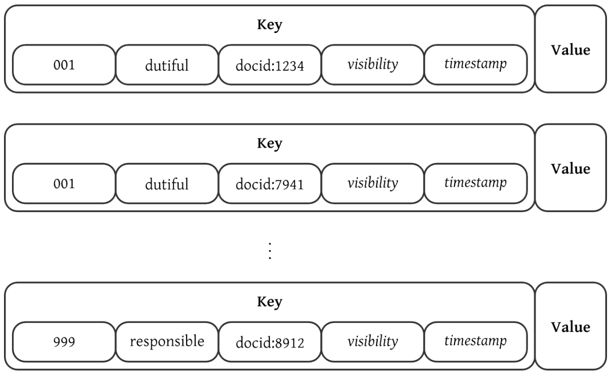

Fig. 4. Index Row Key Format

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

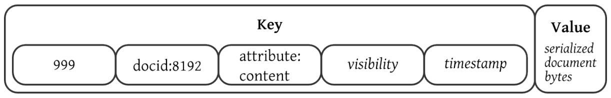

Fig. 5. Data Row Key Format

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 index), 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. 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 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 a geospatial indexing scheme is essentially a way to encode geo-time information in keys so that their natural ordering innately makes spatio-temporal queries 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 point's geohash string representing the geometry with parts of the datetime 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 'yyyyMMddhh' 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. 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'. (See Fig. 6.) 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:

1) Row ID 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 bounds, 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. The random shard can have values between 00 and 99. Therefore, the spatio-temporal bound corresponding to `u` and `201205` is uniformly distributed across compute and storage resources.
2) 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).

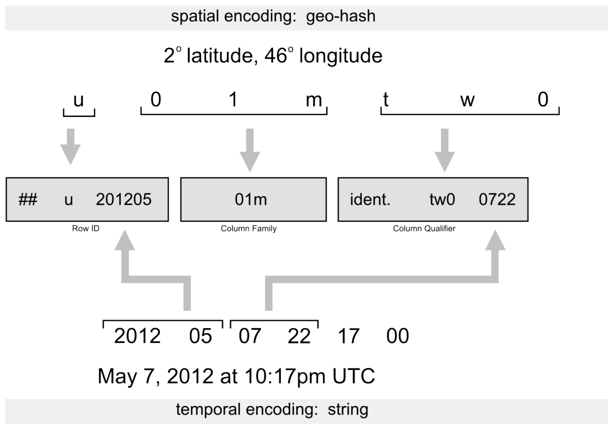
spatial encoding: geo-hash
2° latitude, 46° longitude
u 0 1 m t w 0
## u 201205
Row ID
01m
Column Family
ident. tw0 0722
Column Qualifier
2012 05 07 22 17 00
May 7, 2012 at 10:17pm UTC
temporal encoding: string

Fig. 6. Example of encoding geo-time data into an index entry

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[1].

B. Storing non-point geometries

This index is designed primarily to store and query geo-time data whose geometry consists of a single point, but there is secondary support built-in for non-point geometries. The challenge of storing non-point geometries is that the indexing scheme assumes a single location (35-bit geohash) per stored entry. To accommodate geometries that cover multiple geohashes, 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. This method introduces duplication into the index, so a final, client-side iterator is required to ensure that each qualifying feature's identifier is returned no more than once.

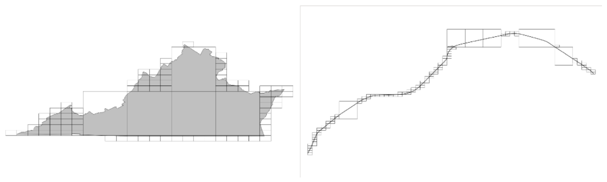

Fig. 7. The (greedy) decomposition algorithm starts with the minimum-bounding geohash that covers the target geometry, and maintains a list of covering geohashes sorted in descending order of "wasted" overlap. For as long as the list contains fewer geohashes than the maximum specified by the indexing scheme, the algorithm decomposes the head (worst) geohash into its child geohashes, and computes their overlap with the target geometry, and inserts these geohashes into the ordered list.

[1]This 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.

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.

C. Query Planning

Computing the results of a query involves building an iterator stack that coordinates an index iterator and a data iterator and optimizes traversal of the underlying storage of the table. As a working example, consider querying for events in New York City in May of 2012. First, the query polygon is converted to its minimum bounding rectangle from which we can determine the coarse geohashes that need to be inspected within the row range. Since New York is contained within the geohash `dr5`, we can limit our query to only those rows with `d` in the row key. Second, the temporal bounds, `201205`, are appended to the geohashes to further refine the row range. Finally, the shard ids are prepended to the row keys at which point we have fully identified every row that must be inspected: `[00-99]~d~201205`. This process effectively narrows in on only the data that must be considered when computing results. The next level of spatio-temporal resolution in the query refines the data traversal by identifying the column families that need to be inspected. In the example query, we need to consider geohashes that are contained within `dr5` thus column families must contain `r5`. During this phase of query processing, the bloom filters enable quick checks to determine if a particular block must be opened at all. Once a full index key has been identified as a candidate result, the index iterator coordinates the data iterator traversal to find the actual data record which is checked against the original polygon and temporal predicate.

The iterator stack is built using a combination of Accumulo system iterators (row and column-family filtering) and custom iterators (geospatial, temporal and attribute filters). Figure 8 illustrates how the iterator stack is built up and how each layer contributes to query processing.

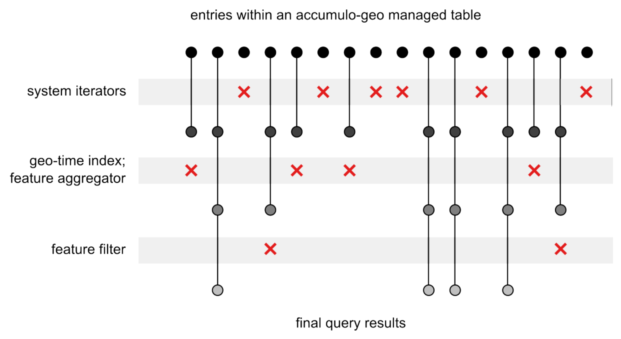
entries within an accumulo-geo managed table
system iterators
geo-time index; feature aggregator
feature filter
final query results

Fig. 8. The spatio-temporal iterator stack. Entries are presented to the iterators in lexicographic sort order. The system iterators comprise the first part of the stack and filter the majority of data from consideration in query processing. The geo-time index in turn filters the output of the system iterators. Finally, the feature filter applies any attribute predicates to the data records identified as candidates by the first two iterator layers.

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.

IV. Performance Results

A. Test data

For our testing, we chose the Global Database of Events, Language, and Tone (GDELT)[12]. This dataset consists of over 200 million geo-political events from across the globe coded using CAMEO codes [3] 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 non-uniform spatial distribution. In figures 9 and 10, we can see this distribution across a map of the world and by geohash.

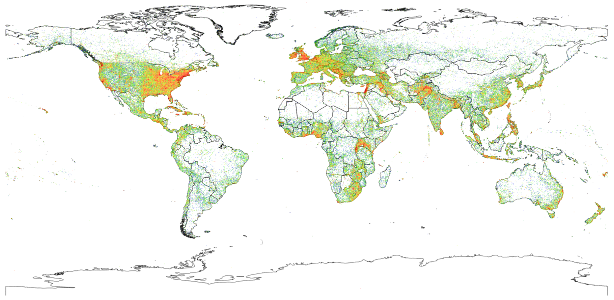

Fig. 9. GDELT event counts aggregated per 21-bit geohash (colored by quantile)

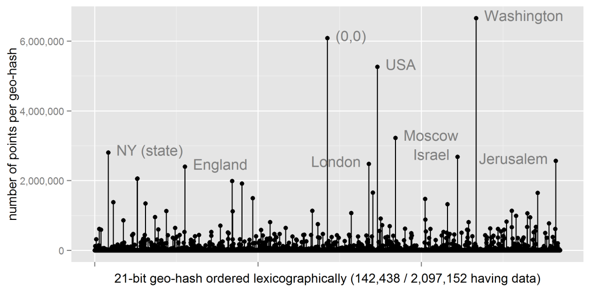

Fig. 10. Distribution of GDELT events across 21-bit geohashes

B. Queries

There were four broad categories of experiments 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 Fig. 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 polygon was chosen because it contains at least 100 data points.

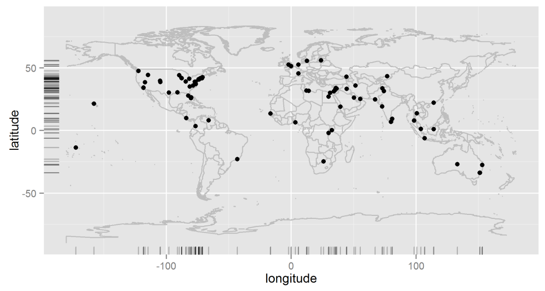

Fig. 11. Distribution of our standard test-set of polygons sampled from the larger GDELT database. Most of these polygons represent specific 35-bit geohashes, though there are a few larger polygons included in the set.

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 Fig. 12.

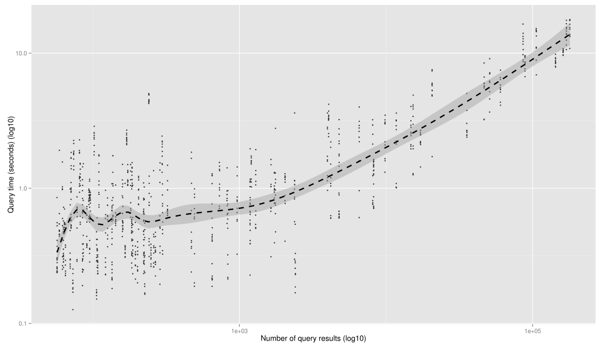

Fig. 12. Response times for a standard set of global query-polygons. This is a log-log plot to highlight the lower end of the scales.

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.

period size	mean responses	mean throughput (records/sec)
day	436	2,019
week	1646	8,034
month	4564	32,988

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

polygon	interval	attributes
London	-	-
London	August 2012	-
London	-	EventBaseCode='010'
London	August 2012	EventBaseCode='010'

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'`, corresponding 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 investigate 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

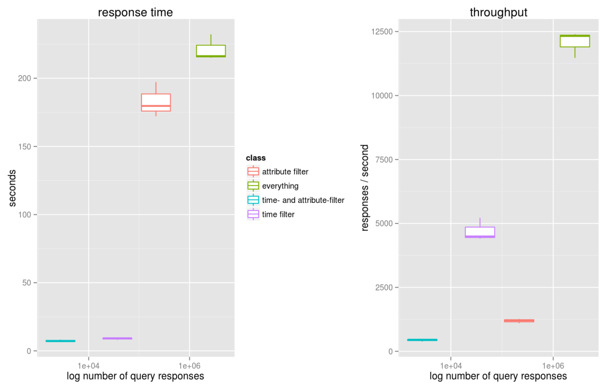

Fig. 13. Attribute filtering queries

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

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.

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

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 tablets 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.

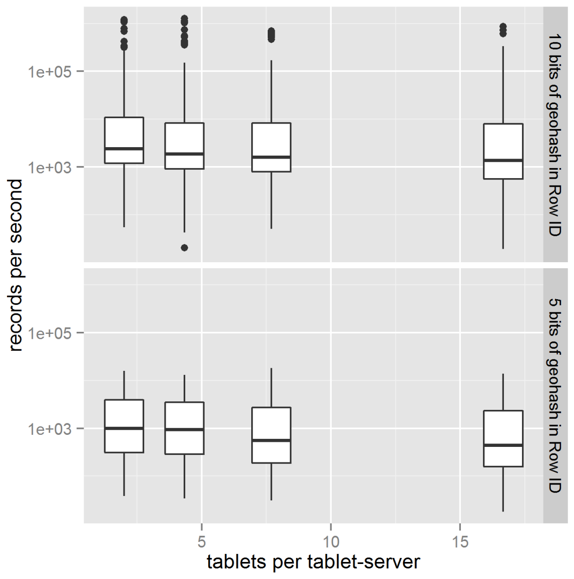

Fig. 14. Scaling experiments.

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 constrains the row to cover 1/32 of the Earth, whereas two drops that fraction to 1/1,024, making each row much more selective. If there were no other effects, it would be easiest to put the entire geohash into the row ID, but this causes an explosion in the number of explicit row-ranges that must be queried, and increases the likelihood that the query-planner will bin them all together into a nebulous blob that is so general that the queries take a very long time to filter out the false positives. It is also relevant that most of these query polygons are based on 35-bit geohashes, meaning that most of those queries will use exactly one combination of characters in the leading two positions. (A larger query polygon, or one that crosses high-level geohash boundaries, will draw hits from geohashes that may vary in their second – or even first – character position.) This tendency increases the discriminating power of the second geohash character, though it is specific to the data set and the queries run.

The main results from this experiment are that this indexing scheme will, as a general rule, perform faster as there are more tablet servers available; that the number of shards should be kept relatively low with respect to the number of running tablet servers; and that – while the default index-schema format works reasonably well as-is – elements of the schema can, and should, be tuned for each data set. We are actively researching how to accomplish this automatically as we will mention briefly in Section V-A.

V. Results Analysis/Conclusion

In traditional RDBMSs, R-trees (and other spatial indices) are external to the data and must be updated when new features are added. Such updates are computationally intensive and affect performance adversely as data size increases. Our geohash-based approach allows us to calculate a key for each piece of data independently. 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 we plan to explore in the future. First, 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 have not directly studied how different queries decompose and their effect on query times. Further in Section III, 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 Section II-B, 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 "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 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. We are investigating a number of refinements to this nearest-neighbor approach, but the existing implementation serves as proof-of-concept that the library is adequate to support a rich variety of geo-time algorithms.

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.

References

[1] 2d Indexes - MongoDB Manual 2.4.4. http://docs.mongodb.org/manual/core/2d/. [Online; accessed 20-June-2013].

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] CAMEO Event Data Codebook. http://eventdata.psu.edu/data.dir/cameo.html. [Online; accessed 20-June-2013].

[4] Michael J. Carey and Donovan A. Schneider, editors. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*. ACM Press, 1995.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In Brian N. Bershad and Jeffrey C. Mogul, editors, *OSDI*, pages 205–218. USENIX Association, 2006.

[6] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[7] ECQL Reference. http://docs.geoserver.org/latest/en/user/filter/ecql [Online; accessed 20-June-2013].

[8] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[9] Apache Foundation. Apache Accumulo Shard Example. http://accumulo.apache.org/1.5/examples/shard.html. [Online; accessed 20-June-2013].

[10] Geohash. http://en.wikipedia.org/wiki/Geohash. [Online; accessed 20-June-2013].

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[12] Philip Schrodt Kalev Leetaru. GDELT: Global Data on Events, Language, and Tone, 1979-2012. *International Studies Association Annual Conference*, April 2013.

[13] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 546–557. ACM, 2002.

[14] J. Kuan and P. Lewis. Fast k nearest neighbour search for r-tree family. In *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, pages 924–928 vol.2, 1997.

[15] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[16] Dimitris Papadias, Yannis Theodoridis, Timos K. Sellis, and Max J. Egenhofer. Topological relations in the world of minimum bounding rectangles: A study with r-trees. In Carey and Schneider [4], pages 92–103.

[17] PostGIS - Spatial and Geographic Objects for PostgreSQL. http://http://postgis.net/. [Online; accessed 20-June-2013].

[18] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In Carey and Schneider [4], pages 71–79.

[19] David Smiley. Lucene solr 4 Spatial Deep Dive. http://www.slideshare.net/lucenerevolution/lucene-solr-4-spatial-extended-deep-dive. [Online; accessed 20-June-2013].