SpatialTable: A Distributed Database for Multidimensional Queries

Daniel Speyer dls2192
Samuel Lee hsl2113
Peiran Hu ph2439

November 30, 2019

Abstract

Both spatial databases and distributed databases have proved useful for a variety of problems. Furthermore, it is possible, albeit inelegant, to store spatial data in a nonspatial distributed database by means of geohashing and this has proved useful as well. We propose a natively spatial distributed database, prototype it, and evaluate its performance. While it is not as performant as the existing best-of-class solution in the most common case, it is superior in several other cases. Finally, we make the claim that these cases are important, and therefore that natively spatial distributed databases are worth pursuing.

1 Introduction

With the explosion of GPS and other location data available from virtually all internet connected devices, coupled with incredibly rich mapping APIs provided by Google and others, applications of all types have been built that use this data for everything from locating the closest restaurant that caters to your late night cravings, to easily splitting the cost of a cab with the closest stranger that happens to be going your way. Applications of this sort have a need to store and quickly query this type of data. However, querying this geospatial data with traditional SQL databases has proved to be very inefficient without the use of pre-generated indexes. Furthermore, it is very difficult to efficiently distribute these types of databases and their additional indexes across multiple database servers, making scalability a difficult proposition at best. There is no solution currently available to efficiently query this type of two dimensional data without the use of workarounds, which can only be put in place if the use of the data is predetermined. With this type of problem in mind we set out to build a database that was built from the ground up to easily and efficiently handle this two dimensional data without any additional indexes. Additionally, we believed that our design would also extend beyond 2-d to n-dimensions in a efficient manner not possible with current databases, and distribute to multiple servers. This multidimensionality would allow our database to solve a range of problems not currently feasible. A discussion on some of these applications are included in section 5. We have created *SpatialTable* with the previously stated goals, and we firmly believe that it provides a solution to the problem of efficiently making multidimensional queries.

2 Related Work

2.1 Distributed Databases

The current standard for distributed databases begins by having a single primary key (usually a string) and arbitrary data. The table is sorted by the key, and chopped into "tablets" of consecutive rows. Which tablet is on which server is stored in a metadata table, which is part of the database and fits this structure nicely. This makes it easy to search by key, or for a range of keys, but to search for anything else requires either a separate index or scanning through the entire table.

2.2 Spatial Databases

There exist many single-node spatial databases, mostly based on rtrees,[1] quadtrees,[2] or similar structures.[3] They are widely used in geographic applications, as a single node can hold a fairly large amount of data. Still, like all single-node systems, they have a cap on their scalability, and they are not robust against hardware failure.

2.3 GeoHashing

Geohashing allows 2-dimensional data to be mapped to a one dimensional key with some locality preserved. Conceptually, a finite fractal "z" shape is drawn over the space, and every point in the space is mapped to the closest point on the fractal. Note that this requires both the range and precision of the data to be known at index-construction time. The distance along the fractal provides a scalar descriptor of the location. This can be used as a primary key in a classic distributed database.[4]

A query in 2-space can then be translated into an approximate query in z-space. Given a box, it is computationally straightforward to find the first and last points along the z-curve that are inside the box. Then all points inside the box must also have z values between those two points. Not all values between those points must be inside the box, but for most queries, most of them will be. Specifically, the expected overhead grows logarithmically in the size of the total database.[5] A database can then scan between the points and filter for being inside the box, with good expected (but poor worst-case) efficiency. This is illustrated in figure 1.

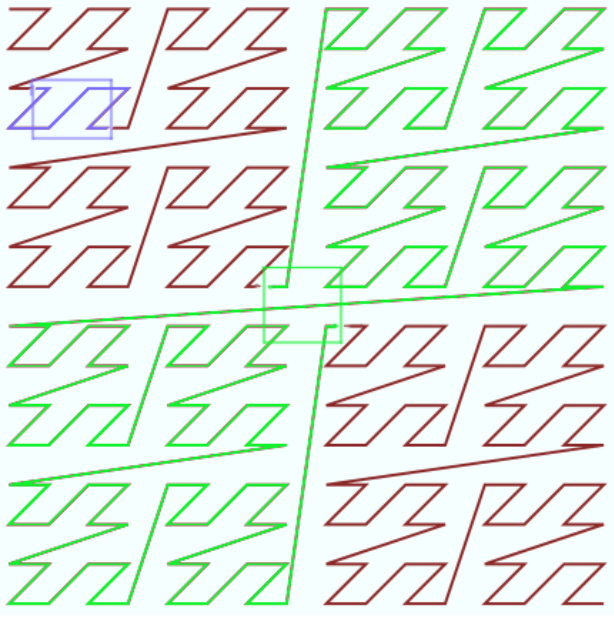

Figure 1: A 4th degree z curve, showing two queries and their respective linearizations. The blue query has only 50% overhead, whereas the green has over 30x

2.4 Hyperspace Hashing

Hyperspace hashing is another multidimensional technology. First, each key attribute is 'hashed' (the reason for the scare quotes will become apparent). These hashes are then treated as axes of a geometric space, which is statically broken into hyperrectangles (called "regions") which are statically assigned to servers.[6]

According to the original HyperDex paper, the system scales poorly with a high number of regions. Since regions grow exponentially with dimension, HyperDex recommends that dimensionality be kept low, and high-dimension tables be divided into 'subspaces' using replication. The system does offer transactions in this replication, though one must worry about the robustness of such a complex system. Also, it requires all the data to be stored once per subspace. Furthermore, queries that specify multiple subspaces only get the benefits of indexing for one, and must scan and filter for the others.

While HyperDex does support range queries, it requires that "objects' relative orders for the attribute must be preserved when mapped onto the hyperspace axis." Since these attributes are not being hashed, but the attribute space is still divided statically, the system is at high risk of hot spots unless the data's distribution is well known in advance.

3 Design

Rather than layering on top of a 1-dimensional database, our design uses arbitrary-dimensional tablets as first-class members, giving full flexibility of dimension, range, resolution and distribution with no scan-and-filter needed for standard queries.

3.1 General Design

Our design is largely inspired by Google's Bigtable.[7] Like Bigtable, we divide our table into tablets, keep our metadata in a table like the original one, and cap the recursion at two levels. Also, like bigtable, we use a general distributed filesystem (hdfs) as our backing store.

Our system is intended as a proof-of-concept, not a production-ready system. As such, we do not support true statelessness as bigtable does, but store data in RAM and write to hdfs eventually. As such, we are not robust against node failures, but the technologies of write-ahead logging and compactions are already well-established, and we would discover nothing new by re-implementing them.

We did consider implementing our system as an add-on to HBase (an existing open source distributed database) but the codebase there was too large and insufficiently documented, so this was not practical.

In a later section, we discuss both what would be necessary to render our system production-ready, and what would be necessary to merge it with an existing database.

Since we want to use our same technology for metadata, and the natural shape for tablets is hyperrectangles (henceforth known as "boxes" for brevity), the keys to our rows are boxes as well. The data associated with the row is an arbitrary binary blob. For our tests, we used strings, but a user is welcome to put protobufs there (as we do for metadata). Supporting bigtable-like columns would again be a practical feature of no research significance.

3.1.1 Tablets

Each tablet has borders (a box) and optionally a list of perpendicular hyperplanes through that box which all entries inside the tablet must cross. For brevity, we refer to these hyperplanes as "lines", which they are in the 2-dimensional case. Since all the lines must be perpendicular, there can be at most as many lines as dimensions. In this all-lines case, all entries inside the tablet must contain a single point. The borders of the box may include infinity or negative infinity. An entry from (1,1) to (2,3) would qualify as inside of a tablet from (0,0) to (5,3) but would not qualify as crossing a line at $\dim_0$=2.

This definition, combined with the splitting algorithm, allows us to maintain a vital invariant: for any possible entry, there is always exactly one tablet that should contain it.

3.1.2 Splitting

When a tablet becomes too large, we split it along one dimension, producing three tablets: 'less', 'crossing' and 'more'. The 'less' and 'more' tablets have smaller borders and the same (if any) lines which must be crossed. The 'crossing' tablet has the same borders, but a new line where the split occurred. Entries are then assigned to the new tablets based on how they relate to the split line in that dimension.

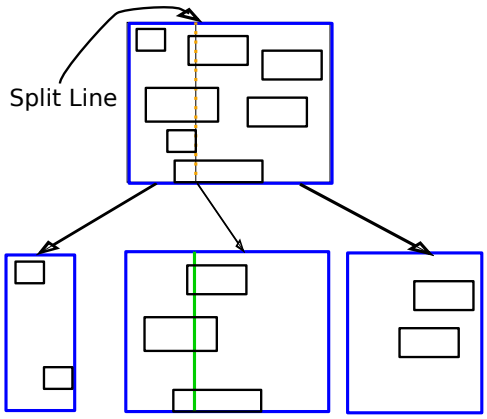

Figure 2: Splitting a 2d tablet

Note that the 'less' and 'more' tablets have never-before-seen borders, and the 'crossing' tablet has the same borders as the tablet which

was just destroyed. This pattern ensures that there will never be two tablets in the same table with the same borders, and therefore that we can safely use the borders as the metadata key.

Finding the split line is a matter for heuristics. The only constraint is that we cannot split a tablet in a dimension for which it already has a must-cross line. Also, it is useless to split a tablet in such a way that all the entries land in the same child tablet. This leaves considerable freedom. Our current heuristic is to take the bounding box of the data actually there, then split it in the widest dimension. To find the point along that dimension at which to split, we take the median after dropping the edges. In practice this works fairly well, but it is neither optimal nor robust against pathological data.

3.1.3 Finding a Single Entry

To find a single entry, we look in md0 for metadata tablets that contain the box, ignoring must-cross lines, then we look in each of them for the tablet that contains the box and whose must-cross lines the box does satisfy. This may seem counter-intuitive, but consider a box which in the relevant dimension ranges from 11 to 12, inside a tablet from 10 to 20, inside a metadata tablet from 0 to 30 crossing 15. This is a perfectly valid arrangement, even though the entry could not be placed in the metadata tablet.

The number of metadata tablets that must be looked at is one more than the number of splits that location has gone through. Since splits are equivalent to a binary tree, the expected number of tablets to search is logarithmic in the total number of metadata tablets. There is no balancing mechanism, however, and with pathological data it would be possible to have to search 2/3 of the metadata tablets.

3.1.4 Conducting a Query

Queries may be of the form 'all boxes within this box' or 'all boxes intersecting this box'. In either case, we must search all tablets that intersect the query box.

3.1.5 Load Balancer

The job of the load balancer process is to distribute the "load" of the overall database among the different tablet servers. We have defined the load in our case to be the amount of rows contained in each tablet. We follow a simple algorithm to determine which tablets to move and it is as follows. First, we measure the load in each tablet, but instead of querying each table and counting the number of rows, we make use of the fact that each tablet is an rtree and we simply obtain the size attribute for each rtree which corresponds exactly with the number of rows. Next, we determine which servers have the max and min loads and what is the target number of rows we would ideally like to move to balance the load. It is important to note however, that we do not move individual rows from server to server, but instead move whole tablets. This means that even though we have a target number of rows we would ideally like to move, we still need to search for a tablet that best matches the number of rows we would like to move. If we find one, then we move it from the max loaded server to the min loaded server. If no such tablet can be found, then we do nothing, and wait for the next iteration to hopefully find a better candidate. In our experience, this worked decently well, since we are also leveraging the fact that we are splitting the tablets that get too large and thus have mostly evenly sized tablets.

3.1.6 Locking

We have a lock on each tablet name, which is held whenever the tablet is in use (including being created or destroyed). We also have a lock on the map from names to tablets that the tabletserver maintains, which is held whenever creating or destroying tablets. These are internal locks within the tabletserver that exist to prevent simulataneous accesses from corrupting the data structures.

There is exactly one circumstance in which a lock is held another is waited on: when splitting a tablet, the original tablet remains locked until the metadata updates are complete. Since

the metadata tree is shallow and acyclic, this cannot produce a cyclic wait nor can a significant fraction of threads be blocked at once, unless there are serious delays in an md0 operation.

This provides a tablet-level linearization guarantee to the user. Since tablets are not a user-visible structure, this amounts to a row-level linearization guarantee.

3.2 Test Infrastructure

Our infrastructure consists of a four-node cluster of virtual machines with Ubuntu 14.04 LTS on KVM. Each VM is itself a datanode on our distributed file system Hadoop File System (HDFS).[8] Our code base is written in C++, using various libraries, and our testing software for comparisons to other database systems was written in Java and Python.

3.3 Dependencies

3.3.1 HDFS

HDFS is a fault tolerant scalable distributed storage component of the Hadoop distributed high performance computing platform, inspired by the Google File System. We chose this file system because it met our reliability, scalability, functionality, and performance goals, and has a very well documented installation and development API in C++.

3.3.2 Boost Geometry

Rather than implement our own spatial data structures in memory, we rely on the Boost Geometry[9] library's rtree implementation to hold each tablet. Inconveniently, this uses templates for the dimension, requiring the table dimension to be known at compile time. We can work around this for most purposes using function pointers, but It does limit the dimensionalities we can support.

3.3.3 Boost Serialization

Since we were already using Boost Geometry to implement our tablets, it was natural to select Boost Serialization as a way to write our data structures to persistent storage, in our case HDFS. The stated goals of Boost serialization of deconstructing an arbitrary set of C++ data structures to a sequence of bytes fit our needs perfectly, however we quickly realized that the stated goal was in fact still a goal, even when dealing with other Boost libraries such as the Boost Geometry. However, we were able to workaround this incomplete implementation and were able to successfully use Boost Serialization to save our data structures to HDFS.

3.3.4 Protobuf

We use Google Protobuf[10] for our wire serialization needs. This is the same library used by HBase and many other distributed systems, so we have a degree of compatibility there.

3.3.5 RPCZ

LibRPCZ[11] is an open-source rpc client/server library using protobufs. It offers clean interfaces for both synchronous and asynchronous RPCs.

Unfortunately, it uses a naive round-robin thread scheduler which can cause it to hang if too many RPCs take place before the first one completes. Special thanks to Dmitry Odzerikho of that project for helping us to understand and work around this bug.

4 Results

4.1 Description of the Benchmarks

We do all our benchmarking using queries, because other operations might not be a fair test. SpatialTable accepts an insertion as soon as the data is in RAM. Other systems might wait for confirmation from the storage layer. That would take considerably longer, but not reflect an advantage of our system.

We generate random data using 5 gaussians, spread uniformly across the range $(0, 1)$ with $\sigma = 0.1$ in all dimensions. A heatmap of the resulting data is shown in figure 4.

We then generate random center points for queries using the same distribution. We select

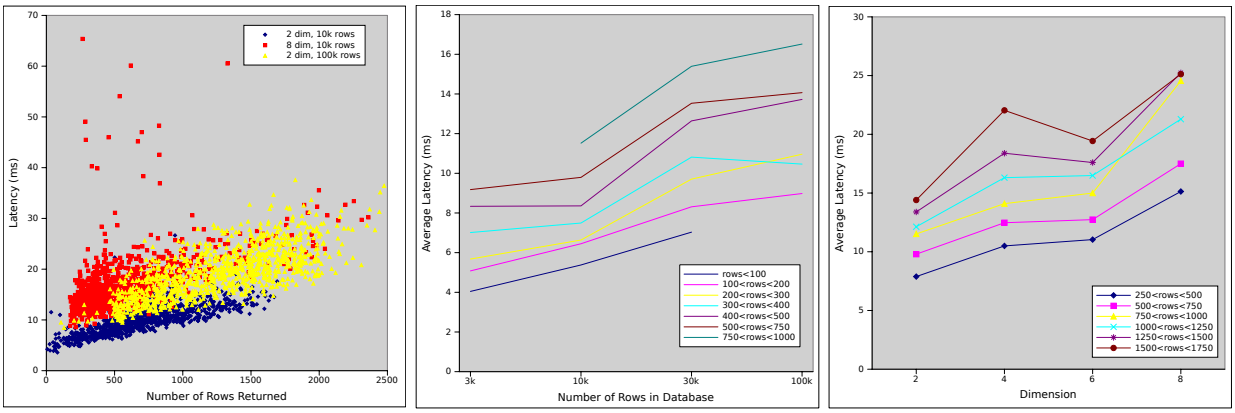

Figure 3: SpatialTable Performance: (a) Latency vs Rows Returned for individual queries, showing 10k rows of 2 dimensions, 10k rows of 8 dimensions and 100k rows of 2 dimensions; (b) Latency vs Rows in Database, averaged and blocked by rows returned; (c) Latency vs Dimensions, averaged and blocked by rows returned

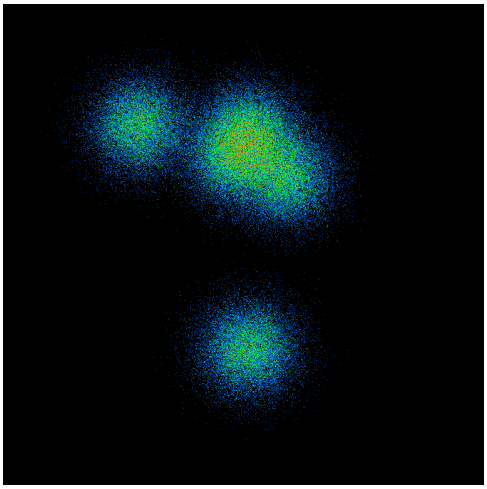

Figure 4: A Heatmap of 100k Random Rows

an expected number of results uniformly from the range $(500, 2000)$ and create a (hyper)square query to return this. However, if the query is in a very low-density region of the map, we limit the size of the query so that it does not enter a higher density region, which produces some queries with significantly fewer results.

4.2 SpatialTable Performance

Drawing from a database of 100 thousand 2-dimensional rows using this distribution, SpatialTable has a mean latency of 19ms ($\sigma = 4.8$) with a 95th percentile of 28ms and a maximum (out of our 1000 test queries) of 38ms.

The number of rows returned had a strong effect on latency, every hundred rows costing an extra 0.7ms ($r^2 = 0.54$).

The number of dimensions had a weaker effect, each dimension costing about 1.4ms ($r^2 \approx 0.8$). This is a smaller effect because an extra 200 rows in a response is far more likely than an extra dimension. Our test at 8 dimensions also showed a significant number of outliers, which our tests at 4 and 6 dimensions did not. This might be an artifact of the testing.

The size of the total database also has a small effect, with a factor of 10 increase costing about 5ms. Both the figures for dimension and database-size effect are highly susceptable to change in how they are measured. The volume of the query box did not effect latency at all.

All of this is illustrated in figure 3.

4.3 Comparison to HBase

HBase is a column-oriented scalable distributed database built on top of Hadoop file system. Tables in HBase are split into regions served by different region servers. If the size of a region is beyond the configurable threshold, master server will handle load balancing by shifting the regions from over occupied servers to less busy servers.

Data in each table are sorted by row-key. A table consists of multiple Column Families (CF), and each column belongs to a specific CF. The

cross point of row and column is defined as "cell" in HBase with its assigned timestamp. These above storage structures make HBase suitable for supporting fast random access (read/write) to huge amount of semi-structured and structured data.

However, HBase does not natively support spatial data processing. Row-key in HBase only supports one-dimentional access to stored data. It is possible to layer geohash,[12] or similar technologies (Hilbert space-filling curve[13] or Grid spatial index method[14]) on top of HBase, but these are not well-supported.

Rather than attempt to install additional layers on top of HBase, we saved geohashing for MongoDB and used HBase's native filtering system: SingleColumnValueFilter. Our experimental result is presented in figure 5. Here, we compared the average latency performance of SpatialTable and HBase based on varied number of rows in database. Naturally, as the growth in the size of rows, average latency of two databases is increasing. However, HBase suffers severely from the increased database size. While, SpatialTable has a almost flat performance. It indicates that SpatialTable will far better outperform HBase with even larger database size.

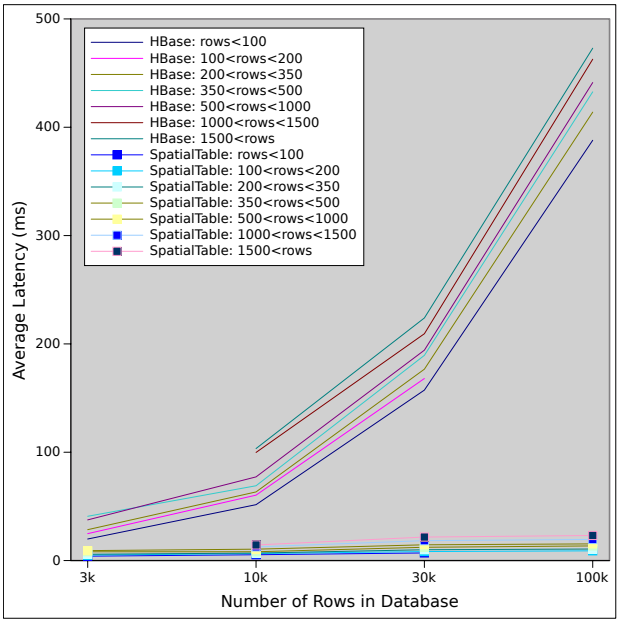

Figure 5: HBase vs SpatialTable at different database sizes, bucketed by rows returned

4.4 Comparison to MongoDB with GeoHash

MongoDB is a popular open-source distributed database with built in geospatial support via geohashing.[15,16] This makes it a natural choice for a comparison. We compared MongoDB to SpatialTable using 100k rows and the same multigaussian distribution as before.

Mongo is approximately 5ms faster than SpatialTable in the median case, but suffers from high outliers. At the 95th percentile, performance is roughly even, and at the 99th SpatialTable is almost 20ms faster. This is shown in more detail in figure 6.

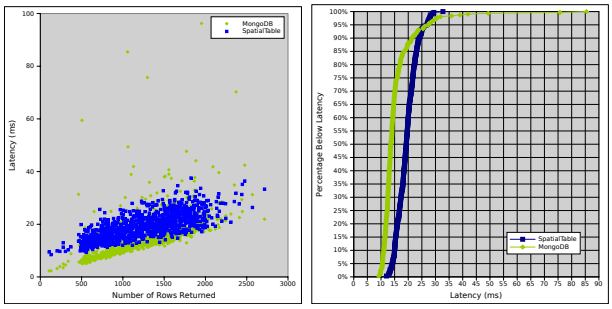

Figure 6: MongoDB vs SpatialTable Drawn from 100k Rows (a) Individual Requests by Rows Returned; (b) Cumulative Histogram for Requests Returning 1000-1500 Rows

These outliers are exactly what we expect from geohashing in general. We also expect that a distribution with many points concentrated below the resolution will cause problems. To trigger this, we reran the test using an power law distribution. Each co-ordinate's base 10 log was drawn from a uniform distribution of (-9,9). We placed the high-density point at the origin to prevent floating point underflow.

With this dataset, we saw a sharp multimodality of MongoDB latencies (see figure 7). The exact nature of this pattern is unclear, but to a first approximation for 38% of queries the geohashing system worked well, while for the rest it hit resolution issues. For queries which returned between 1000 and 1500 rows, only 21% worked well. The exact numbers are not important, as they are an effect of the exact data chosen, but the lesson is clear: geohashing

does not cope well with power-law-distributed data.

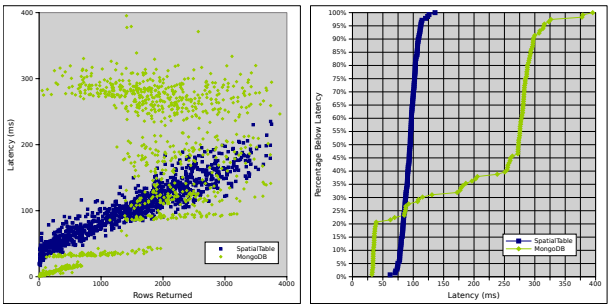

Figure 7: The same graphs as figure 6, but with an power law distribution of rows

4.5 Theoretical Comparison to HyperDex

HyperDex is an implementation of hyperspace hashing – the existing technology truly distinct from geohashing. Unfortunately, we were not able to make it work. Even simple queries produced undocumented errors from deep in its internals.

We can, however, note that to support range queries, hyperspace hashing must use "order preserving hashes". Combined with its static assignment of regions, this ensures it will fail dramatically in the power law distribution case. Not only will its hashing contribute nothing, but it will store the vast majority of data on a single node. If this traffic causes that node to fail, the database will become essentially unavailable.

5 Applications

For most practical geospatial tasks, geohashing is a good solution. It is fast, well-suited to gps data on the surface of the Earth, and robustly implemented. Nevertheless, it suffers from three issues: fixed resolution, strict two dimensionality and high outliers. We believe there are cases in which these issues are important enough to justify the use of a SpatialTable-like solution instead.

5.1 High Resolution

5.1.1 High Resolution Geospatial

The resolution of geohashing is sufficient for human-sized objects and gps-resolution locations. Many "smart city" projects propose higher precision. One approach being prototyped involves sensors a few centimeters in size embedded in sidewalk tiles recording microclimate and traffic information.[17] A more ambitious approach is to access all security cameras and merge them into a single image.[18] In this case, the resolution would be the pixel's size from the camera: highly variable but often submilimeter. Thus far these proposals are limited to cities, but unifying all data across large regions, such as the United States, seems inevitable.

5.1.2 Power Laws in Abstract Spaces

Power law distributions are extremely common in nature for nonspatial properties. For example, websites have power law distributions for both traffic and inbound links (or, when they don't, it's close enough to that for our purposes).[19] If one wanted to keep a database of these and query it by combinations of those, SpatialTable would be well suited. Geohashing would not be an option, both because the highest possible traffic would not be known in advance, and because the most popular websites' traffic exceeds the distinctions of interest among the less popular by more than 2^{26}.

5.2 Higher Dimensional

5.2.1 Geospatialtemporal

The simplest reason to expand beyond two dimensions is to add time. One might create a photo collation system that noted both the location and the date on which the photo was taken. These are, after all, common search criteria and included by default in most cameras' exif data. If the most common search pattern included ranges of both place and time, a three dimensional database would be the ideal backend.

5.2.2 DNA ngrams

Bacterial population studies routinely produce large amounts of data. A single 16S Ribosomal sequence is only a few kilobytes, but each bacterium in the sample contains at least one copy of that sequence. The first task in examining such data is often to match each sequence to similar sequences, either in a reference database or in the rest of the sample. Similarity is defined in terms of alignment, which is very computationally expensive.

One way to make the examinations more computationally tractable is to use ngrams. Take all possible sequences of a given length (there are 4^n) and count how many times they appear in the sequence, including overlapping ones. Then divide the counts by their sum. While there is no theoretical guarantee, empirically similar ngram signatures predict similar DNA sequences quite well, even for small n.[20]

One option would then be to create a $4^n - 1$ (subtract one because the values sum to 1, so the last one provides no information) dimensional table in SpatialTable and store all the sequences in it. Then for each new sequence, consider only a small box around it to compute precise alignment scores.

5.3 High Reliability

There are circumstances under which the 99th percentile performance is more important than the median. For example, consider a control system for a swarm of mobile robots. A general increase in database latency harms efficiency, but a single timed out request has the potential to cause physical collisions, damaging the robots or their cargo.

For a less dramatic example, consider a web service with numerous backends. Each user query results in parallel queries to all the backends. The user-perceived latency is determined by the slowest backend. Even if the queries are in sequence, the anomalously slow ones will account for a large fraction of the total time spent.

There are also psychological reasons to focus on bad-case latencies for any user-facing system. Intermittent very bad experiences cause more user aggravation than consistently mediocre ones.[21] Furthermore, if a handful of users have very bad experiences, those users are likely to be the most vocal, shaping the service's reputation. Perhaps it is for these reasons that Amazon measures all latencies at the 99.9th percentile.[22]

6 Production Readiness

6.1 As a Stand Alone

Converting SpatialTable from a proof-of-concept to a deployable system would require several changes, all of which can be copied with minimal changes from Bigtable or other existing tablet-based distributed databases.

6.1.1 Write ahead Logging

At present, writes to SpatialTable complete when the data is in RAM of the relevant tabletserver, and the data will be written to permanent storage when the tabletserver unloads that tablet. This is clearly not the robustness standard production databases aspire to.

So long as we maintain the principle that all data being served is in RAM in rtrees, managing logging is straightforward. Each write can be appended to a log as well as updated in RAM. The log would then be loadable as easily as a saved tablet. When actually unloading a tablet, we could order the log for optimal loading (something we do not currently do).

Note that in this case the log might contain insertion and deletion events for the same rows. These could be removed in periodic compactions, or we could just wait for unload.

6.1.2 Catching Dropped Tablets

At present, we assume all tablet management operations succeed. To be production-ready, we must handle scenarios in which the tablets loaded, the metadata table and the tablets in persistent storage disagree.

The tablets in persistent storage must be the ultimate authority. The tablets loaded can always be lost in a server crash, and the metadata can be reconstructed from the tablets but not vice verse.

The balancer already makes regular surveys of which servers have loaded which tablets, so it seems well-placed to check for errors. At the same time that it sends those RPCs, it can also read all metadata tables and check all persistent storage.

If the balancer notices a tablet with no metadata, it can simply instruct the least loaded server to load that tablet. That tabletserver will then create the necessary metadata.

If the balancer notices a tablet which is not loaded on the server the metadata says it should be, it should instruct that server to load it. If the server is already in the process of loading, unloading or splitting the tablet, the lock on the tablet name will be held and the tabletserver can return an appropriate error code.

If a tabletserver continues to return error codes for an excessive time, or times out the load or list request, the balancer can kill it. The tablets will now be held by a nonexistent tabletserver.

If the balancer notices a tablet which is held by a nonexistent tabletserver, it can erase that metadata line and issue a load as if the tablet had no metadata.

No tabletserver should ever hold a tablet which the metadata does not list as belonging to it. If the balancer ever observes this, it should kill the tabletserver and file a bug report.

6.1.3 Discovery and Partition-Resistance

Several steps in the dropped tablet process assume the balancer can always talk to all servers, and that there is exactly one balancer. These invariants must be enforced. The classic solution for this is paxos. Maintaining a solitary canonical balancer is a straightforward application. For ensuring connectivity, either a system of tabletserver locks can be used as bigtable does or the tabletservers can directly depend on receiving messages from the balancer. In the latter case, the tabletservers should check paxos on startup to find the address of the current balancer and then inform it of their existence. If a tabletserver does not hear from the balancer for some time (perhaps ten seconds) it should recheck paxos to see if the balancer has moved, and introduce itself to the new balancer if necessary. If it still does not hear, it should kill itself.

As a side effect, either the tabletserver locks in paxos or the introductions to the balancer will provide a way to find a list of all active tabletservers. This is currently hardcoded. In addition to being used by the balancer, this list can be used to find a server to send table-create requests to.

6.1.4 Handling Metadata Timeouts

At present, when splitting a tablet, a tabletserver first performs the split, then removes the old tablet from the metadata, then inserts the new ones. These metadata operations involve sending RPCs to whichever tabletserver is holding the relevant metadata tablet. If those RPCs fail or timeout (perhaps because the metadata tablet was moved at exactly the wrong time) the tabletserver prints an error message and continues, causing database corruption. Several changes are needed to make this robust.

First, failures should be retried after reasonable backoffs. This still does not guarantee success, but it greatly decreases the frequency of failures.

In order to ensure a valid table can always be reconstructed, the order of operations becomes quite complex:

- Create the new tablets
- Write a note to persistent storage indicating that the new tablets aren't real
- Write the new tablets to persistent storage
- Atomically replace the note in persistent storage with one indicating the old tablet isn't real
- Remove the tablet from the list of tablets loaded

- Remove the tablet from the metadata table
- Delete the old tablet's persistent storage (this step could be moved to an asynchronous garbage collector)
- Add the new tablets to the metadata table
- Add the new tablets to the list of tablets loaded

This ensures that there is always one correct table structure which can be reconstructed from persistent storage, and that the tablets will never be seen as dropped (unless the server crashes, and they actually are).

6.1.5 Tablet Naming

At present, tablets are addressed using textual names which are themselves human-readable descriptions of their boundaries. This is very convenient for debugging, but risks name collisions for small tablets. The human-readability is probably worth keeping, but a robust system would need a textualization that guaranteed different strings for different IEEE-754 values.

6.1.6 Security

In terms of security there are a few threats that are most common to the integrity of the database, in no particular order they include, excessive privilege abuse, privilege elevation, denial of service, weak authentication, backup data exposure, and weak database communication protocol vulnerabilities.[23] These have all been previously addressed in commercial database and we can certainly learn from their implementations.

In the case of authentication, we believe we can make use of industry standard methods to hash passwords in a secure store, and also implement strict password requirements. To prevent excessive privilege abuse, we would have to make sure that our database can operate in a way that our users receive only enough privilege to access only what they must. This means that we will not default to giving all users complete access to all database data, and implement different levels of access to allow finer granular control of privileges. Again, there are industry standard practices that have benefitted from years of research and development, that we would try to leverage in our implementation. The same care would go into preventing denial of service attacks, and the safe backup of database data.

6.2 As Part of an Existing Database

Implementing our database on top of an existing project would certainly require us to gain intimate knowledge about the inner workings of their data structures and methods. It would certainly be a considerate time commitment that was too long for us to undertake this semester, and thus inspired us to develop our own solution. However, there are some aspects of typical implementations that are well known and established that we could leverage, particularly the writing of the database log and compaction to persistent storage. It is definitely feasible to use for example the way HBase writes their log and compacts their data, and have the same procedures write our data structures and log to persistent storage. Again, although it seems that there are certain aspects of currently available database systems that we can leverage, overall our design is sufficiently different that would hinder a total integration without a substantial rewrite of our codebase.

6.2.1 Per-Table Metadata

A bigger difference between SpatialTable and traditional distributed databases is that SpatialTable has separate metadata tables for each data table. This is necessary because tables can be of varying dimensions. Even tables of the same dimension cannot easily share metadata tables as there is no spatial equivalent to prepending table names to keys.

Finding relevant tablets is a spatial-specific task that would simply involve spatial code, but every other task that uses metadata (such as iterating over all tablets to ensure they are loaded) would need to be altered to support both

traditional and per-table metadata.

Also, a traditional distributed database only needs a single md0 tablet whose location bootstraps all access. A single hostport can be stored in paxos or other expensive storage without trouble. Permitting this for every table is probably acceptable, but the costs should be monitored. If they prove too high, another layer of indirection is needed. This comes at a latency cost, but perhaps caching can alleviate it.

6.2.2 Protocol

The protocols spoken by existing distributed database are not designed for spatial data. All bitstring primary keys will need to be replaced with lists of pairs. If the protocol was designed for extensibility, this may be possible with minimal disruption.

7 Roles

Peiran Hu was responsible for testing stuff about HBase and MongoDB. She figured out the storage mechanisms and query methods in HBase and MongoDB. She wrote the test cases about 2-D spatial data queries for Starbuck shops through Java Client API for both of the databases.

Sam Lee built all of the infrastructure needed to implement our database, and also installed all of the databases used for comparison. He also wrote the basic HDFS access code, tablet balancer process, and some testing Python code.

Daniel Speyer did the basic design; wrote the framework, fundamental operations, tablet-finding code, splitter and client; and devised the test datasets. He also provided general support to the other team members in their tasks.

References

[1] Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching.* ACM SigMOD. 1984

[2] Finkel and Bentley. *Quad Trees: A Data Structure for Retrieval on Composite Keys.* Acta Informatica. 1974

[3] Samet. *Applications of Spatial Data Structures.* Addison Wesley. 1990

[4] Morton. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing.* IBM Technical Reports. 1966

[5] Tropf and Herzog *Multidimensional Range Search in Dynamically Balanced Trees.* Angewandte Informatik. 1981

[6] Escriva, Wong and Sirer. *HyperDex: A Distributed, Searchable Key-Value Store.* ACM SigCOMM. 2012

[7] Changs et al. *Bigtable: a distributed storage system for structured data.* OSDI. 2006

[8] Shvachko, et al. *The Hadoop Distributed File System* Proceedings of the 2010 IEE 26th Symposium on Mass Storage Systems and Technologies (MSST) 2010

[9] Schiling, Boris *The Boost C++ Libraries* XML Press 2011

[10] Google. *https://github.com/google/protobuf* 2015

[11] Samet, Nadav *https://github.com/thesamet/rpcz* 2015

[12] Dimiduk, Nick, et al. *HBase in action.* Shelter Island: Manning. 2013

[13] Li, Qingcheng, et al. *Optimizational Method of HBase Multi-dimensional Data Query Based on Hilbert Space-Filling Curve.* P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on. IEEE. 2014.

[14] Zhang, Ningyu, et al. *HBaseSpatial: A Scalable Spatial Data Storage Based on HBase.* Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on. IEEE. 2014.

[15] *MongoDB Manual: 2d Index Internals.* http://docs.mongodb.org/manual/core/geospatial-indexes/ 2015

[16] *MongoDB Manual: GeoSpatial Tutorials* http://docs.mongodb.org/manual/tutorial/build-a-2d-index/ 2015

[17] Lange. *The "Connected Boulevard": Taking the Fast Lane to the Internet of Everything.* Cisco Press Release. http://blogs.cisco.com/government/intelligent-communities-global-blog-series-the-connected-boulevard-taking-the-fast-lane-to-the-internet-of-everything 2013

[18] Calavia et al. *A Semantic Autonomous Video Surveillance System for Dense Camera Networks in Smart Cities.* Sensors. 2012

[19] Penneck et al. *Winners don't take all: Characterizing the competition for links on the web.* PNAS. 2001

[20] Sun et al. *ESPRIT: estimating species richness using large collections of 16S rRNA pyrosequences.* Nucleic Acids Research. 2009

[21] Dornic, Stan, and Tarja Laaksonen. *Continuous noise, intermittent noise, and annoyance.* Perceptual and Motor Skills 68.1 1989

[22] DeCandia et al. *Dynamo: Amazons Highly Available Key-value Store.* Symposium on Operating Systems Principles. 2007

[23] Horwath, Jim *Setting Up a Database Security Logging and Monitoring Program* SANS Institute Reading Room 2012