Demonstration of the TrajStore System

Eugene Wu
eugenewu@mit.edu

Philippe Cudre-Mauroux
pcm@csail.mit.edu

Samuel Madden
madden@csail.mit.edu

ABSTRACT

The proliferation of GPS devices has led to a substantial interest in location based services. In particular, modern vehicles can generate an incredible amount of drive data. However, current storage systems are not optimized for storing and querying such large spatial-temporal data sets. In this demonstration, we show the performance of the TrajStore system, a dynamic storage system optimized for quickly accessing data in a particular spatial-temporal region. In particular, TrajStore uses a novel adaptive indexing technique that dynamically adjusts itself to co-locate spatially close trajectories on disk, as well as a number of compression techniques in the storage layer that significantly reduce access time for a given index cell. In this demonstration, we will store a set of real world taxi cab drive traces in TrajStore, and users will be able to query the data through a map based interface.

1. INTRODUCTION

With the advent of cheap GPS and wireless devices, there is increasing demand in applications that support location based services. Such services typically provide users with information related to a geographical region, such as nearby stores, traffic conditions, etc.

Over the past two years the Cartel project at MIT has been collecting driving data from 30 taxi cabs and 15 individuals' cars throughout the Boston Area. In this time we have gathered 245 million GPS points from tens of thousands of drives, amounting to over 68,000 hours of driving. Drives are created after a driver stops the engine, or stays in the same location for an extended period of time. The types of queries we are interested in are aggregate queries over large spatial-temporal regions e.g., for counting how many cars drove through downtown Cambridge along Massachussettes Ave between 5PM and 6PM.

The classic approach to indexing drives is the use of an Rtree [3, 4, 7, 6]. Rtrees are optimized to access arbitrary multidimentional data by indexing an object's bounding rectangle. This does not work well with long trajectories, which tend to have large bounding rectangles and enclose many trajectories. Furthermore, Rtrees only store rectangles and not data, so each matching bounding box requires a separate IO. To address the problem with long trajectories, related work has proposed segmenting trajectories in order to reduce the overall area of the bounding box [6, 5]. These systems assume one page access per segment, and aim to split the trajectory in order to minimize the number of page accesses. However, such systems still required one seek per trajectory, limiting performance over large, multi-trajectory datasets.

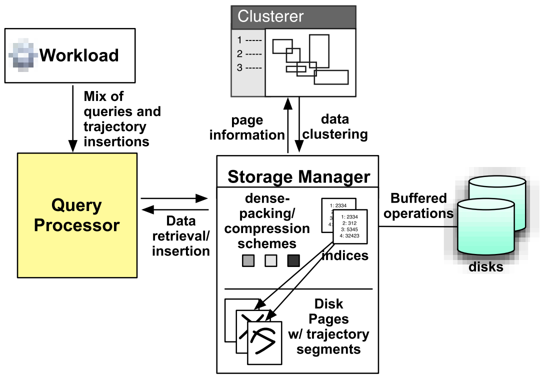
The second common approach uses a grid structure. Grid systems [2, 5] typically find an optimal cell size, split the trajectory at the cell boundaries, and store the segments in each cell together on disk. Given a dataset and query size, this approach works very well. The limitations, however, are that the grid is statically defined and the cell size is typically calculated offline. This means that this scheme is not adaptable to variable query workloads nor continuous inserts into the system.

In light of these limitations, we have developed a storage system that is optimized for spatial-temporal queries over large sets of trajectory data. TrajStore is designed to split trajectories into segments, and co-locate segments that are geographically near each other. It stores segments from each region in temporal order, and then dense packs the data within each region into pages on disk. We utilize an adaptive spatial index to find relevant regions for a spatial query, and a sparse temporal index on the pages to lookup trajectories by time. Using this method, we are able to minimize the number of superfluous pages that are read from disk and avoid the large number of IOs in the RTree based approach. In fact, when compared to existing RTree [6] or grid based approaches on real driving data, TrajStore performs more than 8 times faster.

In this demonstration we exhibit the performance of the TrajStore system with an exploratory visualization of taxi trajectories in the Boston area. In particular, we show the enhanced performance of using the adaptive index, and show how our storage system can be used to answer interesting aggregate queries. These aggregation techniques enable the visualization to accurately visualize millions of data points without loss of usability or performance.

2. SYSTEM ARCHITECTURE

TrajStore is a storage system optimized for storing and querying trajectories. Trajectories consist of a time ordered vector of (lat, lon, t) along with metadata such as a trajectory id, a car id, accuracy measures and other values. A query defines a spatial region r, and a temporal range t, and returns the collection of trajectory segments. Each segment is a sub-sequence of a trajectory that is enclosed in r, and has one or more points in the temporal predicate t.

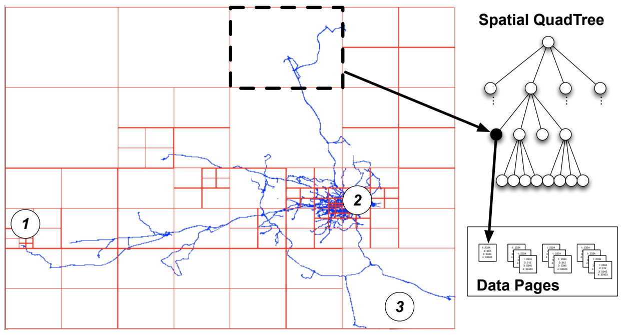

Workload
Mix of queries and trajectory insertions
Clusterer
page information
data clustering
Query Processor
Data retrieval/ insertion
Storage Manager
dense-packing/ compression schemes
indices
Buffered operations
disks
Disk Pages w/ trajectory segments

1: The TrajStore Architecture

The basic architecture of TrajStore is shown in Figure 1. When a trajectory is inserted, the spatial index determines which cells the trajectory segments are stored in, and the storage manager efficiently lays the segments on disk on a per cell basis. Similarly, when a query is posed to the system, the index looks up the intersecting cells, and reads the segments in the data pages on each cell.

The indexer uses a new adaptive spatial indexing technique that we have developed. The key insight is to divide the space into a set of optimally sized rectangles and store the segments in each rectangle together on disk in order to minimize the number of pages when reading a given spatial region. This approach dynamically merges and splits the rectangles as new data enters the system, and as the query workload changes.

TrajStore tries to reduce disk seeks and data transfer by compactly co-locating neighboring trajectory segments on the same disk page. Our primary structure is an adaptive quad tree index that merges and splits cells in order to minimize the number of disk transfers. The leaves of the quad tree point to series of pages that store the data.



2: Our index structure, which recursively splits cells in four in order to minimize the number of accesses to the storage layer.

Figure 2 shows the index of a few hundred trajectories in the Boston area. On the left hand side, point 1 shows an instance where the cells are split to isolate a few segments. Point 2 shows a dense area that is heavily split because accessing any region of space is costly and the overhead of loading data from suplerfluous areas must be avoided. Point 3 exhibits a scenerio in which there is a large sparse area with a handful segments that fit on one page. In this case, splitting is suboptimal because it increases the number of disk pages.

As car data continuously arrives, we are constantly inserting new trajectories into TrajStore. In order to optimize disk accesses, the quad-tree index automatically merges and splits cells according to a cost function. The following function estimates the cost of accessing data from a particular cell, in terms of number of pages (hence the ceiling operation):

$$Cost_{cell}(q) = \frac{(cell_w + q_w)(cell_h + q_h)}{area} \left\lceil \frac{\sum_{i=1}^{4} p_i}{pageSize} \right\rceil .$$

Where q is the query rectangle, p1...p4 are the number of points in each of the 4 quadrants of the cell, and pageSize is the size of a disk page. When we insert a segment into a cell, the value of the updated cost function determines which one of three actions will occur:

- **Split** This occurs when the cost of a cell after an insertion is greater than the cost if it were replaced by its 4 child cells. In this case, the cell will split, and the segments in the cell will be segmented again to fit into the child cells.
- **Merge** A less common case is when the cost of the parent cell is less than the sum of the costs of the current cell's siblings. In this case, the current cell and its 3 siblings push their data to their parent, and remove themselves from the index. This scenerio typically occurs in sparse regions, where it is cost effective to group adjacent segments onto a single disk page rather than maintaining multiple nearly empty cells.

- **Append** By far the most common case, this is when the cell capacity is stable and the segment is simply appended to the last page in the cell.

Finally, after we identify the modified pages, we update their temporal indices, which reflect the smallest and largest timestamp for a given page. This metadata is used to quickly search through all the pages in a cell and locate segments that satisfy the temporal component of a query.

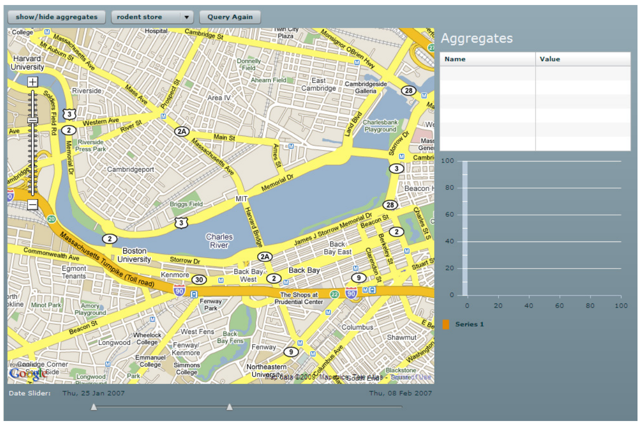
We also provide mechanisms to support adaptations to changes in the query workload by slightly augmenting the cost function. Rather than assume a static query size q, we use an exponentially weighted moving average query size (QS) for each cell. Whenever a cell is queried, we update the dimensions of QS with respect to the query. In practice, we don't constantly update the QS used by the cost function — it is updated when the cost function of QS is more than epsilon different than the QS maintained by the cell.

The second major component in TrajStore is the Storage Manager, which utilizes two compression techniques to reduce the number of pages to store each cell. The first technique works on a single trajectory basis, whereas the second exploits similarities between trajectories. In both cases, the goal is to eliminate as much redundancy in the data as possible. Some of these techniques have a side effect of being very useful for aggregation queries, as we will discuss later.

The first technique is a lossless delta compression scheme to encode successive space and time coordinates within a trajectory. Rather than storing each latitude, longitude point, we simply store the differences, or deltas, between each successive pair of points. For the majority of the points, we can store the delta in a single byte, while larger delta values may require 2 or 3 bytes. We found a compression factor of 4 when applying this technique to real world data.

The second technique exploits the fact that within a given area, many cars will drive along the same route. We cluster nearly overlapping segments in each cell into cluster groups, and store a single segment per group. This method is lossy because it relies on storing summary segments. The user can configure the clustering algorithm to keep errors within epsilon meters. Because each car that produced a trajectory may travel along the route at a different speed, we extrapolate and delta encode the timestamp values for each segment. Applying the previous delta compression technique alongside this method results in a compression factor of nearly 8.

An interesting side effect of this compression technique is that it effectively performs approximate aggregation over the data. For most visualizations that render trajectories across a very large area, rendering a large number of routes that overlap does not improve the utility of the visualization, and even degrades the performance of the application. Instead, returning summary segments for each group of overlapping trajectories alongside the size of the group is a faster, and equally as useful, alternative. By materializing these aggregates in advance, the system can avoid the cost of performing the grouping during runtime. This property is an optimization that we highlight in our demonstration.



3: Screenshot of main interface

3. DEMONSTRATION OVERVIEW

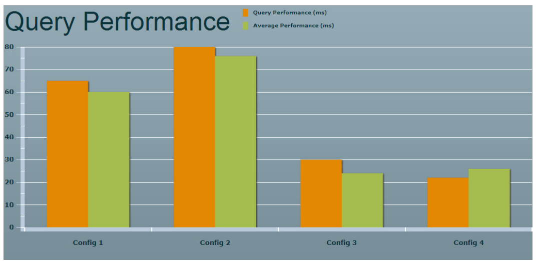
In this demonstration, we exhibit the performance of trajstore through an interactive demo. The demo highlights TrajStore's performance on a large real world data set of taxi cab traces around the Boston region. Users will be able to query the system through a map based interface, and see interesting statistics about the resulting data set and system performance.

3.1 System Setup

We load TrajStore with 890MB of driving data collected from various taxi cabs in the Boston area. This data consists of 68,000 hours of driving, and 245 million GPS points from January 2007 to December 2008. In order to showcase the improvements in our system, we store the data using 4 different configurations — two indexing methods and two storage layouts. The indexing methods are a grid based index, and the adaptive quad tree index described in this paper. The two storage layouts use either delta compression, or delta compression along with trajectory clustering. From our interface, the user can select any of the four configurations and run queries over that particular data store to compare performance across the schemes.

3.2 Demo Interface

The queries that we run in this demonstration are spatial-temporal queries over the boston area. When the user selects a rectangular region on the map, and a region of time using sliders, the interface dispatches the spatial-temporal query to



4: Screenshot of query performance interface

the selected TrajStore backend which evaluates the query and returns a list of matching segments and other metadata and statistics. The result set populates the interface in two ways. First, the cells that intersect with the query rectangle and the resulting segments are overlayed on the map. Second, the statistics and aggregate information are listed and graphed in the right panel. The following are a few example queries that lend themselves well to our system:

1. "How many cars drove across the Harvard bridge in the past year?"
2. "What does the density distribution of drives look like in Harvard square over the past week?"
3. "How did the average speed in the MIT area vary in the past month?"
4. "What is the average speed across a particular segment of road?". (*Useful in [1]*)
5. "How long does it take to get from Logan Airport to the Back Bay?". (*Requested by the Boston police*)

Figure 3 is a screen shot of the proposed query interface. In addition, there will be a screen that displays the recent and average query performance for each of the data store configurations. As participants use the demo, the interface shown in Figure 4 updates the upper chart with the running average performance, while the lower chart displays the most recent query's performance.

4. REFERENCES

[1] M. G. Baik Hoh, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *MobiSys*, 2008.

[2] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data. *GeoInformatica*, 12(2):143–168, 2008.

[3] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.

[4] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proceedings of VLDB*, pages 395–406, 2000.

[5] V. Prasad, C. Adam, C. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets With SETI. In *Proceedings of CIDR*, 2003.

[6] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proceedings of VLDB*, pages 934–945, 2005.

[7] Z. Song and N. Roussopoulos. Seb-tree: An approach to index continuously moving objects. In *Proceedings of MDM*, pages 340–344, 2003.