TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets

Philippe Cudre-Mauroux, Eugene Wu, Samuel Madden
MIT CSAIL
{*pcm, sirrice, madden*}*@csail.mit.edu*

***Abstract*— The rise of GPS and broadband-speed wireless devices has led to tremendous excitement about a range of applications broadly characterized as "location based services". Current database storage systems, however, are inadequate for manipulating the very large and dynamic spatio-temporal data sets required to support such services. Proposals in the literature either present new indices without discussing how to cluster data, potentially resulting in many disk seeks for lookups of densely packed objects, or use static quadtrees or other partitioning structures, which become rapidly suboptimal as the data or queries evolve. As a result of these performance limitations, we built TrajStore, a dynamic storage system optimized for efficiently retrieving all data in a particular spatiotemporal region. TrajStore maintains an optimal index on the data and dynamically co-locates and compresses spatially and temporally adjacent segments on disk. By letting the storage layer evolve with the index, the system adapts to incoming queries and data and is able to answer most queries via a very limited number of I/Os, even when the queries target regions containing hundreds or thousands of different trajectories.**

I. Introduction

The rise of GPS and broadband-speed wireless devices has led to tremendous excitement about a range of applications broadly characterized as "location based services". These applications will provide users with information that is targeted and personalized to their location, whether it be nearby stores, friends, traffic conditions, etc.

To build these services, it is necessary to collect tremendous amounts of data about users' activities and movement patterns in different locations. In the context of our CarTel telematics infrastructure (see `http://cartel.csail.mit.edu/`) we have been focusing on road-usage, with a goal of reporting on traffic data as well as allowing individual users to browse and mine their driving patterns to detect inefficiencies, recommend alternative routes, find carpools, or detect ailing cars. For the past four years we have been continuously collecting speed, position, altitude, as well as a variety of sensor data, including accelerometer traces and data from the on board diagnostic system (called OBD-II, standard in all US cars since 1996) from a collection of 30 taxi cabs and 15 individual users' cars in the Boston metropolitan area. At this point, our database consists of about 200 million GPS readings, representing tens of thousands of drives and more than 68,000 hours of driving.

Currently we store driving data in Postgres, using the PostGIS spatial extensions to index individual drives in an R-Tree index. This is adequate for storing and retrieving data about a single drive but is extremely slow when trying to retrieve data about or compute aggregates over a particular geographic region, as there are often thousands of drives passing through even relatively small (a few square miles) areas, and retrieving each drive incurs a disk seek to fetch the geometry of the drive. Our experiments show that R-Trees can be orders of magnitude slower than the methods we develop in this paper.

Example applications that need the ability to query many trajectories simultaneously include finding the data about all drives passing through an intersection to aggregate traffic statistics for that region, extracting all drives that go from the airport to some part of the city (to compute, for example, taxi tolls – something we have been asked to do by the taxi commission in Boston), and finding all drives that go from one location to another for purposes of computing average delays or finding the best routes.

Due to the performance limitations of the existing data structures, we built TrajStore, a system optimized for efficiently retrieving all of the trajectories in a particular geo-spatial/temporal region. Although there is a large amount of related work on indexing trajectories, most of the current approaches ultimately perform a disk-seek per trajectory lookup, which can substantially impair performance in our settings for the same reasons R-Trees do not perform well. See section II for more detail about related work; our experiments (Section VI) show that our system is about 8 times faster than one state of the art approach [1].

Unlike most existing systems which simply index geo-spatial data, TrajStore is a storage system designed to segment trajectories and co-locate trajectory segments that are geographically and temporally near each other. It slices trajectories into sub-trajectories that fit into spatio-temporal regions, and dense-packs the data about each region in a block (or collection of blocks) on disk. It uses an adaptive multi-level grid [2] over those blocks to look up data in space and a sparse index in time to answer historical queries (which can be formulated as hypercubes.) In this way, most queries can be answered by reading just a few blocks from disk, even if those blocks contain data from hundreds or thousands of trajectories. In addition to this basic formulation, we make four primary contributions:

1) We describe an adaptive storage scheme that chooses the size of the spatial cells in our quadtree index based on the density of data, the expected (or observed) sizes of queries run over the data, and the page size of the system.
2) We describe how our scheme can adapt the clustering of data as new points are inserted or deleted or as the workload changes, and how it maintains an optimal index at all times by splitting / merging cells recursively.
3) We describe a compression algorithm to compress the trajectories in the cells using an approximate trajectory encoding scheme, which accepts a user-specified error threshold. Our method achieves compression ratios of

7.7::1, leading to overall performance gains of 2–3x.

4) We demonstrate that TrajStore can retrieve results from our real-world data set 8 times faster than existing approaches based on segmenting trajectories and storing them in a clustered R-Tree.

Before describing the details of our system architecture and indexing scheme, we first describe the differences between TrajStore and related systems in more detail.

II. Related Work

There has been substantial related work on storing and querying spatio-temporal data.

The "classic" data structure for indexing moving objects and trajectories is the R-Tree [3]. Unlike TrajStore, R-Trees do not *per se* cluster data and are optimized for accessing arbitrary spatial objects, rather than large numbers of overlapping trajectories. Of course, it is possible to attempt to physically co-locate (cluster) objects in the same R-Tree rectangle together on disk (indeed, in our evaluation of R-Trees we found this was necessary to get anything close to reasonable performance on our data set.) Even so, as R-Trees consider nested bounding rectangles to index the objects, it is very likely that if there are many trajectories passing through a small area, there will be large overlaps in these bounding rectangles, resulting in many I/Os to answer any query.

There have been many optimizations to R-Trees for spatio-temporal data, including TB-Trees [4] and SEB-Trees [5]. TB-Trees are optimized R-Tree indices with special support for temporal predicates. They also do not deal well with very long trajectories that tend to have very large bounding rectangles, and can include a high number of I/Os per lookup. SEB-Trees segment space and time, but are not specifically designed for indexing trajectories. Research on TB-trees and SEB-trees does not explicitly discuss how to cluster data, and both are non-adaptive (i.e., they do not reorganize previously added pages as new data arrives.)

To address the concern with very large trajectories, several systems have proposed segmenting trajectories to reduce the sizes of bounding boxes and group portions of trajectories that are near each other in space together on disk. Rasetic *et al.* [1] propose splitting trajectories into a number of sub-trajectories, and then indexing those segments in an R-Tree. They propose a formal model for the number of I/Os needed to evaluate a query, and use a dynamic programming algorithm to minimize the I/O for an average query size. TrajStore also includes an algorithm for optimally splitting trajectories, but physically clusters those trajectory segments rather than just indexing them. We also show how to efficiently maintain this clustering in the face of insertions and deletions. We explicitly compare against Rasetic *et al.*'s approach in our experiments; we chose them as a point of comparison because their paper [1] shows that they dominates earlier approaches like SEB-Trees and TB-Trees for the kinds of trajectory queries we are running.

SETI [6] also advocates a segmentation-based approach like TrajStore. It segments incoming trajectories into sub-trajectories, groups them into a collection of "spatial partitions", and then runs queries over just the spatial partitions that are most relevant to a given query. Like TrajStore, SETI's approach is on-line, in the sense that it allows new data to be added dynamically. The principal differences between TrajStore and SETI are that: 1) the SETI paper does not describe how the size or geometry of partitions are selected, or whether it changes as inserts occur, which is a key contribution of TrajStore, and 2) SETI does not discuss trajectory compression. Furthermore, the SETI paper provides very few details about how the algorithms and encoding schemes work, such that we cannot compare to the SETI approach. We contacted the SETI authors and they were unable to provide an implementation. However, we compare against conceptually similar approaches (see the *Grid* approaches in Section VI.)

PIST [7] focuses on indexing points rather than trajectories making it directly incomparable to our approach. PIST is similar in spirit to TrajStore in that it attempts to optimally partition a collection of points into a variable-sized grid according to the density of the data and query size using a quad-tree like data structure. Unlike TrajStore, PIST is off-line (i.e., it does not adapt to new data being added dynamically). It also does not compress data. Finally, its load performance is slow – in Section 5.6 of [7], the authors report that PIST takes about 900 s to index 2.5 million points; TrajStore is able to load a 75 million point data set in 900 s on a comparable machine, making TrajStore about 30 times faster.

A number of other systems, such as STRIPES [8], use a dual transformed space to index trajectories. While such indices are very compelling when indexing the future positions of moving objects, they are known to be suboptimal for answering historical queries [9].

Clustering and compressing trajectories is very important in our context because of the very high redundancy generated when sampling the position of urban vehicles at high rates. Spatial clustering has been extensively studied in [10], [11], [12]. These approaches focus on generic methods to extract cluster information from large collections of ad hoc data points. Our clustering problem is more specific, since we deal with series of points ordered in time. Also, we take advantage of the fact that the underlying model for our trajectories are road segments, which we use to directly detect similar trajectory segments within each index cell.

Much of the work related to compressing geo-spatial data is focused on fitting raw data points to an underlying road network [13], [14], [15] that form a piecewise polynomial approximation of the trajectory. As such, this work requires as input a road map, which TrajStore does not require. Cao and Wolfson's approach [14], though dependent on a road map, is the most similar to TrajStore. Cao and Wolfson store all trajectory segments that are within some ϵ distance from a road segment as a list of time deltas encoding when and how fast the segment was traversed. In contrast, TrajStore finds segments that are within some ϵ distance of each other and encodes these segments as a canonical trajectory and a list of time deltas. Finally, work such as [16], [17], [18] focus on single trajectory compression by constructing representative trajectories. The algorithms typically remove points while keeping the compressed trajectory within a user specified error bound. In Section V, we describe an efficient window algorithm (as in [18]) that is modified to take into account time-stamps, and show how to combine this algorithm with our trajectory clustering techniques.

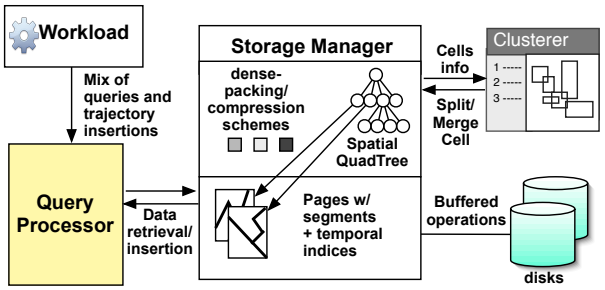

Fig. 1. The TrajStore Architecture.

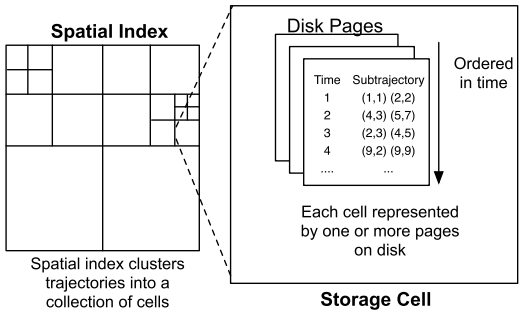

Fig. 2. Basic storage structures to represent data on disk.

III. Overview of Approach

In this section, we discuss the overall design of TrajStore and the basic approach for processing queries and updates. The architecture of TrajStore is shown in Figure 1. The query processor receives a stream of queries as well as new trajectories to store. A query $Q(T, r, t)$ over a table T, a spatial region r (in the form of a rectangle), and a temporal range t returns a set of *sub-trajectories* that lie within r during t. The sub-trajectories are portions of larger trajectories clipped to the rectangle r.

Tables have a schema of the form $(f1, \ldots, fn, Tj)$ where $f1, \ldots, fn$ are metadata pertaining to a trajectory Tj. The metadata may include, for example, a trajectory id, a user identifier specifying who collected the data, some measure of accuracy, and so on. This metadata is specified at the time table T is created. New trajectories are added via a call to $insert(f1, \ldots, fn, Tj)$. A trajectory Tj is simply a vector of (x, y, t) or (x, y, z, t) of tuples, ordered by time-stamp t.

A. Storage Structures

TrajStore storage structures are optimized for spatial queries over specific regions with relatively large time bounds, rather than finding just one or a few trajectories that pass through a region at an exact point in time. For this reason, our storage is primarily organized according to a spatial index, with temporal predicates applied on the data retrieved from this spatial index, as we expect spatial predicates to be generally more selective than temporal predicates.

The basic storage layout of TrajStore is shown in Figure 2. Data in TrajStore is stored by the spatial index and clusterer, as described in Section IV. TrajStore employs several indexing algorithms, all of which work by segmenting trajectories into sub-trajectories, and then clustering spatially co-located sub-trajectories into *cells* on disk. The primary new spatial index that we propose is an adaptive quadtree, where each cell in the quadtree corresponds to a collection of pages on disk that contain the trajectory fragments in that cell. The metadata regarding the bounds and location on disk of the cells fits in memory (as shown in Table II, it consists of a few hundred rectangles for hundreds of megabytes of trajectory data.)

Each cell consists of one or more disk pages, which are the fundamental unit of transfer between the file system and TrajStore; in our experiments, we set the page size to 100 KB. New pages are allocated from a free list; by keeping pages relatively large, we limit the relative fraction of time spent seeking between pages when reading a cell from disk.

TrajStore is primarily optimized for append only storage; trajectories are assumed to arrive in insertion order, and new sub-trajectories are simply appended to the most recent page allocated for each cell in the spatial index. We believe this is a reasonable assumption as trajectories are generally collected from GPS devices and should not change or need to be updated after the fact. To support removal of data, TrajStore associates a "deleted" bit with every sub-trajectory as discussed in Section III-D.

As sub-trajectories are added to a cell, new pages may be allocated. We associate a sparse *time-stamp index* with each cell that simply stores a *start-time-stamp* and *end-time-stamp* for each page in the cell, making it easy to find the subset of pages in a cell that overlap a given time predicate.

Each cell can be compressed using a collection of compression schemes, as described in Section V. The main idea with our compression approach is to eliminate spatial redundancy which arises when many trajectories traverse approximately the same path through a cell.

TrajStore maintains an in memory buffer pool of pages that have been recently read or updated. Pages are written back using an LRU cache management policy. TrajStore maintains a standard sequential write-ahead log of unflushed updates to pages to facilitate crash recovery.

Finally, TrajStore maintains a *trajectory index* which, for each trajectory, maps to a list of cells containing the sub-trajectories of that trajectory.

B. Processing Queries

To process a given query $Q(T, r, t)$, TrajStore first performs a lookup on the spatial index to retrieve the cells overlapping r. The system then uses the time-stamp index to find all relevant pages that overlap t within a cell. Finally, it fetches the pages from disk and linearly scans the sub-trajectories that satisfy the spatial and temporal predicates to extract those that overlap r. This produces a set of *(sub-trajectory, time-stamp)* pairs. The system then assembles sub-trajectories from the same trajectory together in time-stamp order to produce the final query results.

C. Handling Insertions

Each new trajectory is processed by the clusterer, which possibly re-organizes one or more of the cells in the index, splits the trajectory into sub-trajectories, updates the cells on disk (allocating new pages as needed), and finally updates both the temporal indices attached to the pages and the trajectory index. The details of cell creation and management are specific

to the clustering algorithm and are described in Section IV. To prevent concurrent queries from seeing insertions, each query's temporal predicate is constrained to discard time ranges of trajectories inserted after the query began.

D. Deletions and Space Reclamation

To delete a trajectory Tj, the system uses the trajectory index to find all of the sub-trajectories of Tj. It then sets the deleted bit for each of those sub-trajectories to 1 and removes Tj from the trajectory index.

Additionally, TrajStore maintains a "deleted count" for each disk page that indicates the number of deleted sub-trajectories on that page. When a page's fraction of deleted trajectories goes above some threshold, the page is added to a reclamation list. Periodically a reclamation process runs, processing a cell at a time. For each cell that has a page on the reclamation list, it looks at each of the pages needing reclamation and writes a new version of the page to disk discarding the deleted sub-trajectories. When processing a cell, the reclamation process will whenever possible merge together lists of pages that are partially empty, updating the temporal index on the cell as it runs.

To prevent the reclamation process from interfering with concurrent modifications, each cell has a "reclamation lock" on it that can be held concurrently by many inserters or queriers but must be held exclusively by the reclamation process.

IV. Spatial Index and Clusterer

In this section, we describe the new spatial indexing technique we have developed. The key idea behind this technique is to segment space into a series of optimally sized rectangles for retrieving large numbers of spatially related sub-trajectories. Our approach dynamically adapts the index as new data is loaded or as the query workload evolves.

A. Motivation for a New Technique

We devised a novel, adaptive and iterative algorithm to co-locate spatially related segments on disk. As discussed in Section II, there already exist several approaches for segmenting trajectories. However, all previous clustering or indexing techniques either require a large number of independent I/Os when retrieving data from a very dense region (e.g., [1], [7], [19], [20] etc.), or suggest uniform, grid-like partitioning schemes (e.g., [6]).

Both classes of techniques have drawbacks in the setting we consider. Splitting individual trajectories [1], or relying on standard storage systems to retrieve the segments [7] incurs a number of disk seeks proportional to the number of segments touched, which is inappropriate for large datasets residing on disk due to high seek costs (see Section VI). Splitting algorithms for small query sizes create a very large number of bounding boxes, which lead to unnecessarily large indices.

Using fixed grid index to split trajectories and co-locate sub-trajectories on disk also has limitations. Query resolution is directly dependent on the size of the cells making up the grid. Big cells are inadequate for smaller queries, as they contain many segments that do not intersect with the query. Smaller cells, on the other hand, require more disk seeks to retrieve a given spatial region and generate a higher number of split points that need to be inserted in addition to the original points in order to reconstruct the trajectories. The optimal grid cell is difficult to determine, as it depends both on the exact spatial extents of the trajectories and the query load. Thus, the optimal grid size changes with the query load or as more trajectories are added to the system (see also Section VI). One of the key ideas in TrajStore, which we present in the following section, is to employ an adaptive technique to determine and maintain index cells optimized for the retrieval of large numbers of sub-trajectories, even when new trajectories are inserted or when the workload evolves.

B. Optimal Cell Size

As with many data warehousing applications, the time taken to answer historical queries on large trajectory data sets is largely dominated by disk operations, since most of the time is actually spent fetching data from disk. As discussed above, TrajStore strives to limit disk seeks and data transfers by compactly co-locating neighboring sub-trajectories on the same disk page. Hence, one of the issues we have to tackle is determining which segments should be co-located on the same page. The answer to that question is not straightforward as it depends simultaneously on the workload, the spatial extent of the trajectories, and the page size.

Consider random rectangle queries q of size $q_w \times q_h$ on the spatial plane. The optimal way of answering such queries (assuming the exact coordinates of the query are known a priori) would be to define a bounding box around the segments touched by the query and to co-locate these segments on the same page(s). Our system was built from the start to handle large data pages of several hundreds of thousands of kilobytes in order to enable data prefetching and amortize disk seeks. Hence, we estimate that the query answering cost is directly proportional to the number of pages accessed to answer the query (we show that this is indeed the case in practice in Section VI):

$$Cost(q) \sim \#pages\ accessed.$$

For a homogeneous region $cell_w \times cell_h$ with a density $\mathcal{D}$ of data points (observations) per square unit, the query answering cost is thus:

$$Cost_{cell}(q) \sim \left\lceil \frac{(cell_w \times cell_h)\ \mathcal{D}}{pageSize} \right\rceil$$

where $pageSize$ stands for the maximal number of data points that a page can contain (note that we neglect the cost incurred by the additional points needed when splitting the trajectory). The above function is highly non-linear – a cell with no points costs 0 (in practice, keeping track of empty cells can be done without disk access) whereas a cell containing any number of points from one to $pageSize$ costs 1.

For a larger homogeneous region of area $area$, more cells are needed. The cost associated with accessing this area given a random query of size $q_w \times q_h$ and for a given cell size is thus [21]:

$$Cost_{area}(q) \sim \sum_{cell} P(q \cap cell) \left\lceil \frac{(cell_w \times cell_h)\ \mathcal{D}}{pageSize} \right\rceil$$

where $P(q \cap cell)$ is the probability that the random query intersects the cell $cell$. This probability depends on the spatial

extents of both the query and the cell. Clearly, the query intersects a cell when its center falls within the boundaries of the cell. It also intersects the cell when its center falls just outside of the cell, up to a distance $q_w/2$ of the vertical edges of the cell, and up to $q_h/2$ outside the horizontal edges [1]. For an area $area$ and by neglecting border effects that happen at the edges of the region (outside of which queries would not be allowed), the probability of a random query q intersecting a given cell is thus:

$$P(q \cap cell) \sim \frac{(cell_w + q_w)(cell_h + q_h)}{area}.$$

By substituting this probability in the cost expression, we obtain the final cost expression for a query:

$$Cost_{area}(q) = \sum_{cell} \frac{(cell_w + q_w)(cell_h + q_h)}{area} \left\lceil \frac{(cell_w \times cell_h)\ \mathcal{D}}{pageSize} \right\rceil.$$

Our objective in the remainder of this section is to show how to minimize this expression.

C. Multi-Level Grid

The cost formula introduced above allows us to determine the optimal grid size for a large homogeneous area of density $\mathcal{D}$, by considering a series of juxtaposed squares of the same size $cell \times cell$ covering the area. Replacing the summation in the previous expression with the number of cells in the total area, which is $area/cell^2$, and again neglecting border effects, we obtain the following expression for the optimal cell size in a regular grid approach:

$$cell_{opt} = \underset{cell}{\operatorname{argmin}} \left(\frac{(cell + q_w)(cell + q_h)}{cell^2} \left\lceil \frac{(cell^2)\ \mathcal{D}}{pageSize} \right\rceil \right).$$

The above expression points out the limitations of a fixed grid approach. The optimal cell size depends on the density of the region considered. However, large trajectory data sets are highly skewed in the spatial dimension, as most traces concentrate around popular areas (e.g., city centers) and roads, while isolated areas (e.g., forests) are mostly empty. Thus, the optimal grid size compromises by averaging the query retrieval cost over all regions. Even more problematic, the density of the data set evolves every time a new trajectory is inserted. Maintaining an optimal grid size in presence of updates would require repeatedly re-clustering the entire data set, replacing all pages with new pages corresponding to a new cell size, which is unacceptable for very large data sets.

Instead of a fixed grid approach, we develop in the following an adaptive, multi-level grid approach, which recursively splits and merges cells in order to minimize the number of disk transfers. Figure 3 gives an example of the resulting index structure. The information about each cell is maintained in a dynamic quadtree, whose cells point to series of pages storing the data. Our dynamic approach splits regions in order to minimize empty space in the cells containing segments. On the left hand side of Figure 3 (point *1*), for instance, the cells are recursively split in order to isolate a few segments. Cells are also divided in dense areas (point *2*), where accessing any point in space is costly and where any overhead associated with the access of superfluous areas has to be avoided. However, splitting

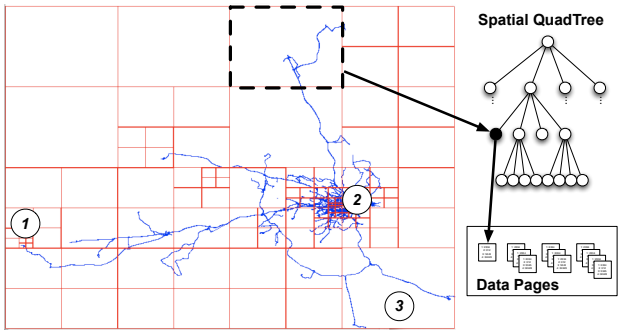

Fig. 3. Our index structure recursively splits cells in four in order to minimize the number of accesses to the storage layer.

cells may also lead to suboptimal partitions, for example for sparse areas containing a few segments only that should be stored together in order to minimize disk access (Figure 3 point *3*) or for bigger query sizes. We give the details of our splitting algorithm below.

D. Index Construction

As our system continually receives trajectory data from a fleet of moving objects, we have devised an efficient, iterative mechanism to maintain an optimal index in the presence of frequent inserts. At any point in time, we keep a spatial index (a quadtree) representing the spatial configuration of the index cells. Each cell is also associated with a cost:

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

where $p_1, \ldots, p_4$ represents the current number of points in each of the four quadrants of the cell. This cost is derived from the general cost expression described above and represents the expected (average) cost for the retrieval of the trajectory segments contained in the cell, given a query q. The first term represents the probability that the cell gets queried, as before. In the remaining cost expressions, we drop the $1/area$ term from this expression as it is a constant that affects the magnitude but not the relative costs of different cell configurations.

We initially start with a set of four empty cells covering the entire area of interest. We split any incoming trajectory according to the spatial index, creating lists of sub-trajectories each of which is spatially contained by a cell. Before writing a sub-trajectory to disk on the pages corresponding to its enclosing cell, however, we split the sub-trajectory one additional time, creating sub-trajectories for each of the (virtual) sub-quadrants of the present cell. Now, suppose we want to insert a sub-trajectory in a cell at level l, where level 0 corresponds to the root of the quadtree, level 1 to its four children, etc. Three different cases can occur:

Split Taking into account both the current number of points $(p_1, \ldots, p_4)$ contained in the cell at level l and the points $(n_1, \ldots, n_4)$ from the new trajectory split by quadrant, we determine the cost associated with the replacement of the current cell by its four children:

$$ChildCost_{cell} = \sum_{i=1}^{4} (cell_w/2 + q_w)(cell_h/2 + q_h) \left\lceil \frac{p_i + n_i}{pageSize} \right\rceil.$$

A new cost smaller than the current cost triggers the $split(cell)$ subroutine (see Algorithm 1), which recursively splits the cell as long as the cost associated with the sub-cells is smaller than that of the parent cells. New sets of pages are created on disk for each of the sub-cells.

Merge Similarly, we compute the cost associated with the replacement of the current cell and its three siblings (which can themselves contain sub-cells) by a new parent cell:

$$ParentCost_{cell} = (cell_w \times 2 + q_w)(cell_h \times 2 + q_h) \times \left\lceil \frac{\sum_{i=1}^{4} n_i + \sum_{i=1}^{4} p_i^{neigh} \ \forall neigh \in desc(par(cell))}{pageSize} \right\rceil$$

where $par()$ retrieves the parent of a cell and $desc()$ all the cells in the sub-tree attached to a given parent (its *descendants*). We call the $merge$ subroutine (see Algorithm 2) whenever the cost associated with replacing the current configuration of cells by a parent cell is smaller than the current cost. This happens in relatively sparse regions, where storing adjacent segments in the same cell is less costly than creating relatively empty cells for each of the segments. $Merge$ reads the sub-trajectories from several cells, sorts the sub-trajectories by time whenever necessary, and produces a merged list of sub-trajectories.

Append For dense regions, adding a few segments to a cell does not typically alter the cost of the cell significantly. When neither splitting nor merging is necessary, we simply append the new data to the page corresponding to the cell and update the values of $p_1, \ldots, p_4$.

We use a cost penalty in practice to avoid splitting or merging cells unnecessarily for insignificant cost gains (i.e., we only split when $ChildCost_c \times (1+\epsilon_{cost}) < Cost_c$ and merge when $ParentCost_c \times (1 + \epsilon_{cost}) < Cost_c$ for a given ϵ_{cost}). All three subroutines maintain a temporal index on the temporal range (smallest time-stamp and biggest time-stamp) of each page. In order to minimize the seeks when fetching data from a cell, both $split$ and $merge$ request lists of pages from the storage manager that are contiguous on disk when writing new cells.

The index constructed in this way is optimal, in the sense that the expected cost of answering the workload is iteratively minimized by splitting and merging cells for every insertion. In environments where inserts are infrequent, we extend the above algorithm and assess the total cost of several split/merge steps for every insertion to ensure that the partitioning converges rapidly.

E. Query Processing

To retrieve trajectory data for a given query, we perform a lookup on the spatial index to determine the cells that intersect the query on the spatial plane. We examine the temporal range of all pages associated with the cells intersecting the query, and retrieve those pages that contain data in the temporal range specified by the query. Finally, we reconstruct the trajectories by merging all sub-trajectories with the same trajectory identifier, and clip the resulting set of trajectories following the exact query bounds to return the results.

Input: A cell $cell$ that will be split
Output: The number of cells $nbNewCells$ that have been inserted into the quadtree to replace this cell

```
int nbNewCells = 0
cell[] children = doSplitInFour(cell)
quadTree.remove(cell)
foreach cell newCell ∈ children do
    if childCost(newCell) * (1 + ε) < cost(newCell) then
        nbNewCells += split(newCell)
    else
        quadTree.insert(newCell)
        nbNewCells++
return nbNewCells
```

Algorithm 1: Algorithm $Split$ for recursively splitting cells.

Input: A cell $cell$ that will be merged with its neighbors
Output: The number of cells $nbMergedCells$ that have been merged and replaced by a new cell

```
int nbMergedCells = 0
cell[] neighbors = desc(par(cell))
cell newCell = doMergeCells(neighbors)
quadTree.remove(neighbors)
nbMergedCells += neighbors.size()
if parentCost(newCell) * (1 + ε) < ∑ cost(cell) ∀cell ∈ desc(par(newCell)) then
    nbMergedCells += merge(newCell)
else
    quadTree.insert(newCell)
return nbMergedCells
```

Algorithm 2: Algorithm $Merge$ for merging cells.

F. Query Adaptivity

Our approach as described thus far easily adapts to the addition of new data—as trajectories are added, grid cells may be split or merged as needed to optimize performance.

We provide additional mechanisms to support query adaptivity. The key observation is that the formula for the cost of a cell is dependent on the query size. One option is to assume an average query size that is used over all time, but this is clearly simplistic. As an alternative, we have implemented a scheme that records an exponentially weighted moving average (EWMA) query size QS with each cell. We update the current EWMA query size $curQS$ (represented as two values, h, w) when a new query $newQS(h, w)$ is processed as follows:

$$curQS = \begin{array}{l} (\alpha \times curQS.h + (1-\alpha) \times newQS.h, \\ \alpha \times curQS.w + (1-\alpha) \times newQS.w) \end{array}$$

where α is an exponential weighting factor. Whenever a cell is queried, we recompute the EWMA as $curQS$ and save the value along with QS. When $|curQS.h \times curQS.w - QS.h \times QS.w| > t$ for some size threshold t, we re-cluster the cell and save $curQS$ as QS. Re-clustering simply involves applying the cost formulas for `split` and `merge` given in Section IV-D with q set to $curQS$. We show the effectiveness of this adaptivity technique in our experiments.

V. Compression

In this section, we describe our method for compressing sub-trajectories stored within a cell. Our approach combines two compression schemes:

1) We use a lossless delta compression scheme to encode successive time and space coordinates within a trajectory.
2) We use a lossy compression scheme to cluster trajectories traveling on nearly identical paths, and store a single representative spatial path for all trajectories along with a collection of time offsets for each trajectory in the cluster.

Our clustering mechanism is particularly effective in the context of our dataset of road trajectories, where different cars drive the same roads and paths over and over again, which leads to a great deal of redundancy. Unlike other compression methods for trajectory data (e.g., [14]), our approach does not rely on an underlying map of road geometry to identify and cluster related trajectories. Although such road maps are available for some parts of the world, they must be kept up to date as roads change and cannot be made to work for pedestrian or off-road tracks, which limits their utility.

A. Single Trajectory Delta Compression

Our delta compression scheme encodes a trajectory of the form:

$$(x_1, y_1, t_1), (x_2, y_2, t_2), \dots , (x_n, y_n, t_n)$$

as a starting coordinate followed by a series of deltas, e.g.:

$$(x_1, y_1, t_1), (\Delta x_2, \Delta y_2, \Delta t_2), \dots , (\Delta x_n, \Delta y_n, \Delta t_n)$$

where $\Delta(x|y|t)_i = (x|y|t)_{i-1} - (x|y|t)_i$. The key idea is that each of these deltas can be encoded using significantly less space than the original value. Because of the difficulty in storing floating point numbers, we encode the deltas using fixed point arithmetic. For the majority of points, we can store each delta in a single byte, with some larger differences requiring 2 or 3 bytes. Overall, delta encoding alone reduces the size of the data set by about a factor of 4, as we show in Section VI.

B. Cluster-based Compression

In combination to delta-encoding, TrajStore employs a cluster-based lossy compression method that encodes multiple sub-trajectories together. The overall idea is to cluster related sub-trajectories in each cell into cluster groups, and store only one trajectory per group. This method is lossy as it relies on storing a summary trajectory for each group, omitting the individual spatial points of each trajectory in the group. Because different trajectories may record traversals of the same roads at different times and rates, we still must delta-encode and store each trajectory's time values. The following describes this algorithm in more detail, as well as additional optimizations.

1) Calculating Distances Between Trajectories: Given two trajectories:

$$t1 = [(x_{11}, y_{11}), (x_{12}, y_{12}), \dots , (x_{1n}, y_{1n})]$$
$$t2 = [(x_{21}, y_{21}), (x_{22}, y_{22}), \dots , (x_{2n}, y_{2n})]$$

we can calculate the distance between $t2$ and $t1$ by considering each point p_{i1} in $t1$ and computing its distance from the corresponding point p_{i2} in $t2$. We find the corresponding point p_{i2} in $t2$ by first calculating p_i's linear distance along $t1$ from p_{11}, and traveling the same distance along $t2$. We then measure the Euclidean distance d between the two points p_{i1} and p_{i2}. Finally, we compute the distance between the two trajectories:

$$traj\_dist(t1, t2) = max(d(p_{i1}, p_{i2})) \ \forall p_{i1} \in t1, \ p_{i2} \in t2$$

We note that $traj\_dist(t1, t2)$ may not equal $traj\_dist(t2, t1)$, as the two trajectories may have a different number of input points. Hence, we report the final distance as:

$$dist = max(traj\_dist(t1, t2), traj\_dist(t2, t1))$$

2) Calculating cluster groups: Cluster groups are formed around a central trajectory t, such that for all other trajectories t' in the group, $dist(t, t') < \epsilon$. We currently select the first trajectory scanned as the central trajectory. Algorithm 3 shows the algorithm for cluster group formation.

Input: A *cell* containing a set of trajectories
Output: A list of cluster groups $\{G_1, \dots, G_n\}$, where all trajectories in a group G_i have $dist < \epsilon$ from each other.

```
1 foreach trajectory t in cell do
2     if t is in a group then
3         continue
      /* t is central trajectory of G          */
4     create new group G = {t}
5     foreach trajectory t' in cell do
6         if t' is in a group then
7             continue
8         if dist(t, t') < ε then
9             G = G ∪ {t'}
```

Algorithm 3: Algorithm for forming cluster groups.

3) Storing Cluster Groups: Once cluster groups have been formed, the compressed cell can be written to disk. For each group, the central trajectory t is written out using delta encoding (as above). For other trajectories, we must record their time vectors. Because these other trajectories may have been sampled at different rates or frequencies than t, we must extrapolate each stored position p in t to the times when each of these other trajectories visited p. This extrapolation is done in a manner similar to the distance calculation given above: for a given non-central trajectory t', for every point p in t, we find the point p' that is the same distance along t' as p is along t. Then, we compute the time $ts_{p'}$ when t' visited p' by looking at the stored time-stamps of the two points in t' that are closest to p', assuming that the object moved at a uniform speed between those points. Finally, for each t', we delta encode and store these $ts_{p'}$ time-stamps.

4) Updating Cluster Groups: When a new trajectory t' is inserted into a cell, we search for a group g with central trajectory t_g such that $dist(t', t_g) < \epsilon$. If we find such a group, we encode t' and store it with the cluster group as above. Otherwise, we create a new group for t' and make t' the central trajectory of that group.

Our results in Section VI show that cluster group compression, combined with delta compression of time-stamps, yields an overall compression ratio of nearly a factor of 8 on our real world driving data.

C. Eliminating Extraneous Points

Because our GPS data is sampled at relatively high frequency, there are often times when vehicles don't move or move along a completely linear trajectory. Hence, as a final compression step, we eliminate extraneous points that can be predicted via linear interpolation from the surrounding points. To describe the algorithm, we first define a function $isSummary(p_i, \dots, p_j)$,

which returns true if the points between p_i and p_j can be linearly extrapolated (see Algorithm 4).

Input: List of points $(p_i, \ldots, p_j)$, ϵ
Output: Returns True if the line between p_i and p_j is an ϵ approximation of points $p_i, \ldots, p_j$. Else, returns False.

```
Compute the line L between p_i and p_j
foreach p_x between p_i and p_j do
    t_x = p_x's time-stamp
    ℓ_x = Interpolated location of a point at t_x along L
    if sqrt((ℓ_x − p_x)^2) < ε then
        p_x is removable
if all points between p_i and p_j are removable then
    return True
else
    return False
```

Algorithm 4: $isSummary()$ Algorithm

To eliminate extraneous points in a trajectory T, we set p_i to the first point in T and find the furthest point p_j along the segment such that $isSummary(p_i, p_j)$ is True. The points between p_i and p_j are removed, p_i is set to p_j, and the process is repeated until p_i is at the end point.

D. Choosing Epsilons

As mentioned earlier, both cluster based compression and extraneous point elimination are schemes that attempt to summarize segments within user specified error bounds. If we set the error bounds for clustering and point elimination to ϵ_1 and ϵ_2, respectively, then the maximum error incurred when using both schemes is simply $2 \times \epsilon_1 + \epsilon_2$. Thus, we can ensure a bound of ϵ for the entire system, by setting the bounds for clustering to $\epsilon_1 = \epsilon/4$ and point elimination to $\epsilon_2 = \epsilon/2$.

VI. Experiments

Having described our scheme for indexing and compressing trajectories in TrajStore, we now evaluate the performance of our approach.

A. Experimental Setup

To analyze the performance of our approach, we implemented and compared several trajectory indexing schemes in the same basic experimental framework.

1) Software Implementation: We implemented TrajStore, including the indexing, clustering, and compression methods in Java 1.6. Our implementation consists of approximately 30,000 lines of code. In all cases, we ran Java configured with 1024 megabytes of memory.

We compared against several approaches (described below). In most cases, we implemented these approaches from scratch using the TrajStore framework, though we did use the open source XXL [22] library for our R-Tree implementation. We experimented with different fanouts for the R-Tree, and finally chose a fanout of 500, which worked best for the approaches that heavily rely on the R-Tree (*ClustSplit* and *NoSplit* below.)

2) Hardware Configuration: Our results were produced on a dual 3.2 GHz Pentium IV with 2 gigabytes of RAM and a 300GB 7200 RPM drive, running Redhat Linux 2.6.16.

3) Data Set: Our primary experiments were with 890 MB (approximately 77 M data points) of driving data from the data set described in the introduction. This comprises 11,014 separate trajectories. Although this data set can fit into the RAM of our machine, the experiments we run are all disk bound as we flushed the buffer pool and OS file system cache (using the Linux 2.6.16 `/proc/sys/vm/drop_caches` method) before each trial. This data set was collected from a fleet of 20 vehicles; however, the techniques described in this paper scale to much larger fleets, whose aggregate size could easily exceed the memory of any modern machine (see Section VI-B.7, which reports scalability results).

This data covers 68,000 hours of driving, from January 24, 2007 to December 3, 2008. The average trajectory is 6,990 GPS readings taken once per second, so the average drive is about 2 hours. Trajectories primarily come from taxi cabs, which tend to have long drives. A new trajectory is created when a car turns off or idles in the same location for a long time. As with most real data sets, this data set is skewed (our taxis drive mostly in downtown and from/to the airport.) We note that this data set is approximately 20 times larger than the data sets used in previous papers [1] on large scale trajectory management, which used about 5 million synthetic data points.

4) Approaches Compared: We compared a number of approaches in our experiments:

Adaptive: The TrajStore adaptive clustering approach, as described in Section IV. The adaptivity (Section IV-F) and compression (Section V) features of our approach were turned off, except for three experiments described below (experiments 3, 4, and 7).

Grid: This approach segments trajectories according to a fixed size grid chosen based on an input query size, and then stores trajectory segments from the same grid cell together on disk using TrajStore's storage manager. It uses a simple spatial index to find the grid cells that can be used to answer a query.

ClustSplit: The method of [1], where a trajectory is split into a number of sub-trajectories based on a dynamic programming algorithm that optimally splits a trajectory for a given query size. The algorithm described in [1] incurs one I/O per sub-trajectory segment retrieved, which yields very poor performance when running queries that retrieve a large fraction of the database, which is our main focus. To improve its performance, we co-locate all of the segments contained within a given leaf node of the R-Tree together on disk, reading them all into memory when a query over any of them is asked. Without this optimization, performance of this approach are about three orders of magnitude slower on our data set because of the huge number of random I/Os required.

NoSplit: This approach stores each of the trajectories in an R-Tree, and performs no splitting. It incurs one I/O per trajectory read. This is comparable to what we use in our current vehicular trajectory database described in the introduction. For most experiments with R-Trees, we chose to use the XXL package over PostGIS (`http://postgis.refractions.net/`) because it offered better performance; however, we also experimented with Postgres/PostGIS R-Trees as a baseline.

CapacityQuad: This approach uses a capacity-bound quad-tree as an index, where each cell in the quad-tree is represented by a single page on disk. Trajectories are segmented according to the boundaries of the quad-tree, and sub-trajectories are stored in each quad-tree cell. A quad-tree cell is split into four sub-cells when its page is full. We stop the splitting process for tiny cell sizes to prevent floating point arithmetic issues.

5) Performance Metrics: We use two performance metrics. The first is the number of I/Os (e.g., disk pages read). In our experiments, we varied the disk page size from 10KB to 1MB. We stored all pages belonging to the same cell contiguously on disk to create large blocks of related data that could be read continuously. Hence, the time taken to answer a query didn't vary significantly with the page size (less than 10% variations for page sizes ranging from 10KB to 1MB). In the following, we use 100KB pages, which we found worked well over a range of experiments, unless otherwise reported. Disk I/O is a good indicator of overall system performance, but does not capture disk seek. For this reason, we also measure the overall running time for each approach.

B. Results

In this section, we present the results of several experiments.

1) Baseline Experiments: We started by running a few simple experiments comparing our system to two well-known approaches. We compared TrajStore to a standard Postgres installation with PostGIS and ClustSplit, our own implementation of the trajectory splitting technique in [1] with clustering of the objects in the same leaf page. We loaded a subset of our CarTel data set representing 250 MB of trajectory data and measured the average time needed to retrieve trajectory segments in 1% of the total area considered. PostGIS with a spatial index on each trajectory required 4.1 s on average, due to the large number of trajectories (and number of points) that are retrieved (many of these points are in fact outside the query rectangle). ClustSplit took on average 9 s to answer the queries, due to the large index it creates (see Table II), which requires a large number of random I/Os to traverse. In contrast, TrajStore took an average of 48 ms to answer the queries, as it needs to read only a few blocks from disk containing the trajectory segments of interest.

2) Experiment 1: Query Size: In the first experiment we ran queries with spatial predicates over 10%, 1%, and 0.1% of the entire region covered by our data set. For this experiment, we queried 100% of the temporal data; we experiment with temporal predicates in Experiment 5 below. For 10% queries, we ran 10 randomly selected queries; for 1% queries, we ran 100 randomly selected queries; for 0.1% queries, we ran 1000 randomly selected queries. Thus, each trial queries approximately the same amount of total data from the system. The same query rectangles were used for each different index. For these experiments, the Adaptive, Grid and ClustSplit methods, which use the query size as input, were given the exact query size for use during indexing. In later experiments we show the robustness of these algorithms when the exact query size in unknown, or when the query size varies during the experiment.

Figures 4, 5 and 6 show the runtimes and number of I/Os for 10%, 1%, and 0.1% queries, respectively. Results are the average number of milliseconds or I/Os per query.

Here, "Adapt." is our Adaptive scheme. It performs the best of all the approaches, both in terms of I/Os and running time. As it groups pages on disk within cells, and keeps cells sized optimally for the query, its overall performance is very good. As expected, larger queries require more I/O and take longer to run, but performance is essentially linear, with about 100x more I/O and running time for a 10% query than a 0.1% query. In general, Adaptive is about a factor of 8 better than the ClustSplit algorithm in terms of total time per query on this data set. The figure also shows the accuracy or our cost model, as the number of I/Os is directly proportional to the time taken to answer the queries, and our model is able to perfectly estimate the number of I/Os (since it has access to the index structure when making decisions about whether to split/merge cells.)

For the grid approach, we computed the optimal grid size by exhaustively iterating through all grid sizes for subsets of our data set. In general, we found that a grid size of between 50% (Grid1/2) of the query size and 100% (Grid 1/1) of the query size works best for gridding. Of course, in practice such exhaustive search isn't possible, but the purpose of this experiment is to show the best these methods can do. Though this optimal grid approach appears competitive here, we show in the next section that it does not deal well with queries it was not explicitly optimized for.

ClustSplit is the optimized version of the technique in [1]; Our optimized version does well in terms of number of I/Os (nearly matching Adapt.) but generally has a large overall runtime because it must perform many random I/Os through the R-Tree when reading pages from disk.

NoSplit, on the other hand, generally reads more pages from disk, because it stores large bounding rectangles that intersect many queries, but sometimes performs better than ClustSplit when many trajectories span more than one disk page. Its performance is worse for small queries as it has to read whole trajectories when only small pieces are needed.

CapacityQuad performs reasonably well, especially for small queries. However, as it writes one page per cell, it ends up doing lots of random I/O to retrieve the results, which hurts its performance. Also, it creates many additional (split) points, which represents a significant overhead both for query processing and storage.

Indexing Time: We also measured the time to build each of the data structures for the entire data set and a 1% query size The results are summarized in Table I. Our adaptive method, while slower that the simple grid-based approach, still performs quickly. Our approach can insert approximatively 100,000 new points per second, which is more than sufficient to handle the moderate update rates in our telematics infrastructure. It does much better than the ClustSplit algorithm, which takes quite a long time to cluster (these numbers are consistent with the results in [1] (Figure 8), which show that for a 5M point data set, the total indexing time was between 5,000 and 20,000 seconds.) CapacityQuad is also very slow as it has to split the trajectories frequently because of all the cells it creates.

TABLE I
INDEXING TIME

Index	Time to build (s)
Adaptive	920
Grid 1/1	380
Grid 1/4	502
ClustSplit 10%	7,174
NoSplit	316
CapacityQuad	8,200

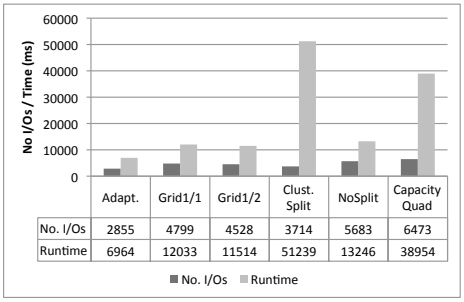

Fig. 4. 10% Queries

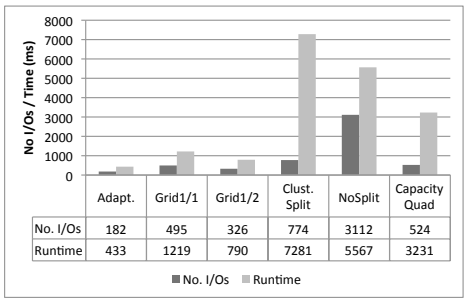

Fig. 5. 1% Queries

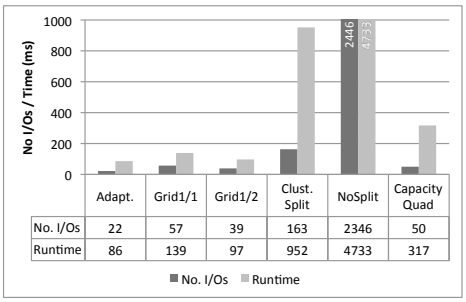

Fig. 6. 0.1% Queries

Index Size: Finally, we measured the size of the data structures and the number of cells created. The results are shown in Table II. The initial data set was 890 MB and about 77 M points. As expected, all methods increase the size somewhat. ClustSplit uses substantially more space as it creates many more segments than the other approaches (since it segments each trajectory individually.) The indices created by Adapt, Grid and NoSplit are several orders of magnitude smaller.

TABLE II
INDEXING SIZE

Index	Size (MB)	Num. cells	Num. split points x1000
Adaptive	935	802	747
Grid 1/1	913	108	313
Grid 1/4	922.1	408	468
ClustSplit 10%	1.5×10^3	14,183	3,521
ClustSplit 1%	2.0×10^3	21,094	5,626
ClustSplit 0.1%	1.5×10^3	30,118	7,965
NoSplit	890	–	–
CapacityQuad	1.5×10^3	15,008	3,780

3) Experiment 2: Sensitivity to Query Size: In the next set of experiments, we look at how well each of our approaches performs when we optimize the indices for a particular query size and then run a set of queries with a different size. This experiment is important because it shows the sensitivity of the clustering and indexing methods—in reality, queries will never be of fixed size, and will vary substantially. Hence, an ideal method would be relatively insensitive to the training query size. Here, we only show results for Adaptive, Grid, and ClustSplit (the two other index construction methods are independent of the query size). In these experiments, we disabled the additional adaptivity mechanisms (Section IV-F) of our Adaptive scheme.

Figure 7 shows the results when we run 10% queries on indices built assuming queries are 1% or 0.1%, Figure 8 shows results for 1% queries on indices built for 10% and 0.1% queries, and Figure 9 shows 0.1% queries on indices built for 10% and 1% queries. These figures show "speedup", which is the ratio of performance of the experiments shown in Figures 4–6 to the performance where the index was built with a non-ideal query size (thus, a result > 1 indicates that it was actually faster/required fewer I/Os when using a non-ideal query size!) The dark horizontal line indicates a speedup factor of 1 (i.e., no performance impact when constructing the index using a different query size.)

These experiments show that the Adaptive approach is relatively insensitive to input query size, with runtime slightly (<5%) faster (for 10% queries on an index built for 1% queries) to about 40% slower (for 0.1% queries on and index built for 10% queries.) Generally, running smaller queries on a index built for larger queries is expected to be more expensive because those queries will have to read substantially larger blocks of data than what the index was tuned for. Our algorithm does well in that case, however, because it tends to create small cells on disk (much smaller than 10% of the entire data set), even when built for queries over 10% of the data.

The Grid approach is unsurprisingly very sensitive to the input query size. It is completely non-adaptive, and performs segmentation based solely on the size of the query, so performance with a different sized query can be as much as a factor of 10 slower.

Finally, the ClustSplit algorithm shows surprising results in this experiment. It is substantially faster (about a factor of 5) to run .1% queries on an index built for 10% queries, although the number of accesses is not that much less. The reason for this has to do with the number of split points generated by the algorithm. These large numbers of split points lead to many more tiny rectangles in our R-Tree built for 0.1% queries. These rectangles tend to be spread randomly across the disk, incurring substantial random I/O. When indices are built for 10% queries, larger cells are written in the leaves of the R-Tree. Though those blocks take longer to read, the index co-locates the pages of these cells on disk, so they are read sequentially, substantially decreasing the number of seeks.

4) Experiment 3: Adaptivity: The next set of experiments measured the effectiveness of our query adaptivity scheme. We initially built clusters for a query size of 10%. We then ran 100 queries of size 1% with our EWMA adaptivity scheme in place and measured after each query the average number of I/Os required to answer a 1% query. We intentionally chose small values for α to increase the convergence speed. The results are shown in Figure 10.

From the graph, we can see that the number of I/Os per query drops as more and more cells adapt to the new query size. Smaller values of α lead to a faster adaptation of the index. Overall for $\alpha = 0.1$, the performance stabilizes after about 100 queries from about 240 I/Os per query to about 180, a performance improvement of 33%.

5) Experiment 4: Real Query Workload: This experiment measures the effectiveness of our scheme on a real query workload taken from a drive management system we built for CarTel. Users can use this system to browse their own historical drives as well as traffic patterns and other trajectory data. Users pose "queries" by panning a map and zooming in or out; we logged these queries over a period of about two months. The resulting workload contains approximatively 5000

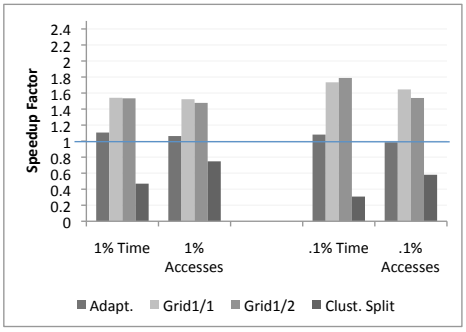

Fig. 7. 10% queries on 1%/.1% indices

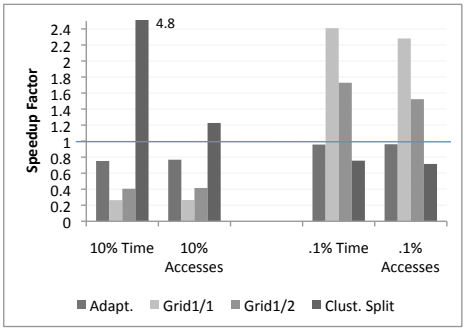

Fig. 8. 1% queries on 10%/.1% indices

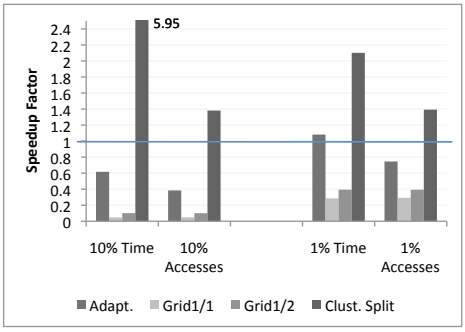

Fig. 9. 0.1% queries on 10%/1% indices

queries. More than 50% of those queries retrieve trajectories in areas representing 0.1% to 1% of the total area considered. A few queries (approximatively 10%) are significantly larger and retrieve more than 10% of the area considered. Figure 11 shows the performance of our adaptive approach compared to the grid approach. The figure shows the cumulative number of I/Os required to answer the queries for 50 steps, where we insert 100 trajectories and then pose 100 queries at each step. Both schemes were initialized using the mean query size of the first 100 queries. Figure 11 includes two versions of our Adaptive approach, one with query adaptivity turned on ($\alpha = 0.8$), and one with query adaptivity turned off ($\alpha = 1$). As can be seen on the figure, our approach clearly outperforms the static grid approach and is able to iteratively adapt to the insert and query workloads. The Adaptive scheme with the EWMA query adaptivity turned on is more effective, but requires longer to index the trajectories (the total indexing time is about 40% slower for $\alpha = 0.8$).

6) Experiment 5: Temporal Predicates: In the fifth set of experiments, we measured the effect of temporal predicates. Our goal was to show that the system is able to reduce query processing time when temporal predicates are included. In addition to spatial predicates of size 10%, 1%, and 0.1%, we added temporal predicates of 50%, 10%, and 1% of the time covered by our entire data. We ran our clustering algorithm with the correct input query size in space. We then measured the ratio of performance of each temporal predicate vs. the performance of the same spatial predicate without a temporal predicate. The results are shown in Figure 12.

The leftmost group of bars shows the performance with 10% spatial queries and varying time predicates. A 50% time predicate runs in about 51% of the time as the query without a time predicate. 10% and 1% time predicates run in about 17% and 8% of the time, respectively. The 10% predicate is still selective, but generally only uses about 60% of the data on each page that is read from a cell, wasting some I/O time. This effect is even more pronounced at 1% temporal predicates, and only a few records are used from each cell that is read. Though the results do not scale perfectly, it is worth noting that we are still reading 175 trajectory fragments from disk in just 408 ms in this case! Results are similar for the 1% spatial predicate case (middle bars). The speedup from temporal predicates is substantially reduced in the 0.1% spatial predicate case (rightmost bars), especially for smaller time ranges. This is because so little data is selected in these cases (we read 8 and 4 sub-trajectories on average for 1% and 0.1% time predicates) that total runtime is dominated by seeks between cells (runtime is 32 ms and 20 ms respectively in these cases.)

7) Experiment 6: Scalability: To confirm that our approach scales with the number of trajectories, we experimented with data sets of various sizes. Table III gives the results for the number of I/Os and the time required to answer 1% queries using our adaptive scheme on data sets ranging from 40 M to 200 M points, corresponding to databases ranging from 470 MB to 2.3 GB approximatively. As can be seen from that table, both the number of pages fetched from disk and the time taken to answer the query scale linearly with the size of the data.

TABLE III
SCALABILITY

	40M	80M	120M	160M	200M
No. I/Os	90	189	275	376	457
Time (ms)	232	448	719	952	1208

8) Experiment 7: Compression and Extraneous Point Elimination: In the final experiment, we measured the performance of our point elimination and compression schemes. For testing compression, we ran both the single trajectory compression algorithm and the cluster-based delta compression algorithm. For these results, we used a page size of 10K. We reduced the page size to 10K because after compressing the trajectories, many of the 100K sized pages would have been underutilized.

TABLE IV
COMPRESSION SIZE

	No Comp.	Delta	Cluster	LZ
Size (MB)	935	233	121	330

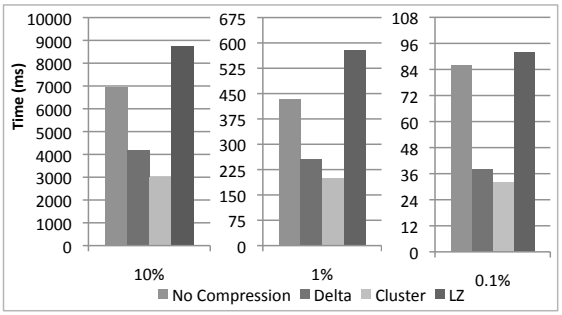

Fig. 13. 10%, 1%, and 0.1% Query Runtime on Compressed Data.

The sizes of the compressed data are shown in Table IV and the query times for the same queries as in Figures 4, 5 and

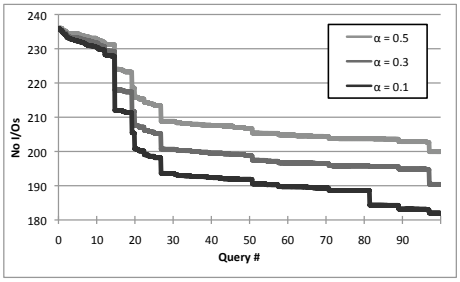

Fig. 10. Query perf. with query adaptation.

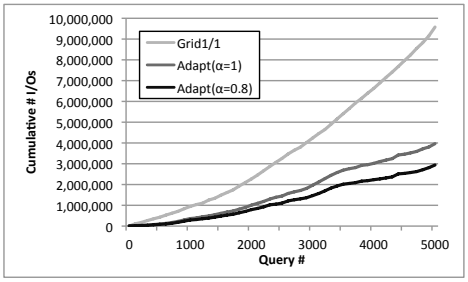

Fig. 11. Query perf. on the CarTel workload.

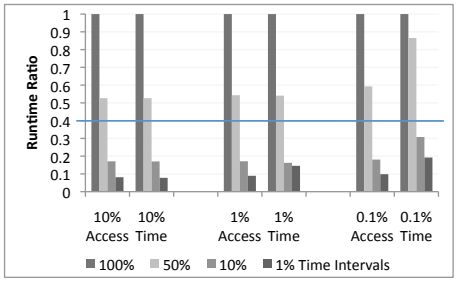

Fig. 12. Query perf. with temporal predicates.

6 are shown in Figure 13. Overall, compression reduces table sizes by a factor of 7.7, and improves performance by a factor of 1.8–2.6. For reference, we also show results (LZ) for a scheme where we use Java's built in LempelZiv compressor to compress blocks as they are written out. We then decompress the blocks before processing them. LZ provides a compression ratio of about 3::1, but actually slows the overall query performance due to decompression costs.

Single Trajectory Delta Compression. Single trajectory delta compression reduces data size from 935 MB to 233 MB, a factor of 4.1. Most points in the trajectories are a very small distance from the previous point, and thus delta compression is very effective. Query performance shows less than a factor of 4 speedup because performance is not purely a factor of storage size. Decompression costs and disk seeks are also significant. Performance gains are greater with smaller queries, since smaller queries lead to fewer seeks and are thus more affected by reduced I/O demands.

Cluster Compression. Here, we set $\epsilon = 1m$ (see Section V-D). We also enable extraneous point elimination with $\epsilon_2 = 0.5m$. Combined, the two techniques reduce the overall size by a factor of 7.8, about half the size of delta compression alone. Extraneous point elimination is responsible for a factor of 2.4 compression by itself. Of the total of 9.7×10^5 total sub-trajectories stored, we ended up with $6.0 \times 10^5 clusters$, each containing 1.6 sub-trajectories on average. The total storage of 121 MB consists of 50 MB of storage for cluster representative trajectories, and 71 MB for time data points.

VII. Conclusions

In this paper, we presented TrajStore, a new scheme for indexing, clustering, and storing trajectory data. TrajStore is optimized for relatively large spatio-temporal queries retrieving data about many trajectories passing through a particular location. Our approach works by using an adaptive gridding scheme that partitions space into a number of grid cells, and adaptively splits or merges cells as new data is added or the query size changes. Our results show that our approach is substantially better—about a factor of 8—than the best existing approaches by avoiding expensive I/Os and dense-packing related data on disk. We also showed that our method is online, in the sense that it adapts as new data is inserted or as queries evolve, and that it is relatively insensitive to configuration parameters such as the input query size. We presented a compression method that eliminates redundancy between trajectories that cover similar paths. Our compression method is able to receive compression ratios of 7.7::1. These results suggest that TrajStore is the system of choice for large scale analytic queries over trajectory data.

VIII. Acknowledgments

This research was supported by the NSF under grant number IIS-0704424 and by Microsoft Research under a Jim Gray Seed Grant.

References

[1] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, "A Trajectory Splitting Model for Efficient Spatio-Temporal Indexing," in *VLDB*, 2005.
[2] K.-Y. Whang and R. Krishnamurthy, "The multilevel grid file - a dynamic hierarchical multidimensional file structure," in *DASFAA*, 1992.
[3] A. Guttman, "R-Trees: a Dynamic Index Structure for Spatial Searching," in *SIGMOD*, 1984.
[4] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," in *VLDB*, 2000.
[5] Z. Song and N. Roussopoulos, "SEB-tree: An Approach to Index Continuously Moving Objects," in *MDM*, 2003.
[6] V. Prasad, C. Adam, C. Everspaugh, and J. M. Patel, "Indexing Large Trajectory Data Sets With SETI," in *CIDR*, 2003.
[7] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander, "PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data," *GeoInformatica*, vol. 12, no. 2, pp. 143–168, 2008.
[8] J. M. Patel, Y. Chen, and V. P. Chakka, "STRIPES: An Efficient Index for Predicted Trajectories," in *SIGMOD*, 2004.
[9] K. Porkaew, I. Lazaridis, and S. Mehrotra, "Querying mobile objects in spatio-temporal databases," in *International Symposium on Advances in Spatial and Temporal Databases*, 2001.
[10] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "OPTICS: Ordering Points to Identify the Clustering Structure," in *SIGMOD*, 1999.
[11] W. Wang, J. Yang, and R. R. Muntz, "STING: A Statistical Information Grid Approach to Spatial Data Mining," in *VLDB*, 1997.
[12] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Effcient Data Clustering Method for Very Large Databases," in *SIGMOD*, 1996.
[13] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk, "On Map-Matching Vehicle Tracking Data," in *VLDB*, 2005.
[14] H. Cao and O. Wolfson, "Nonmaterialized Motion Information in Transport Networks," in *ICDT*, 2005.
[15] N. Hönle, M. Grossmann, D. Nicklas, and B. Mitschang, "Preprocessing Position Data of Mobile Objects," in *MDM*, 2008.
[16] E. Frentzos and Y. Theodoridis, "On the Effect of Trajectory Compression in Spationtemporal Querying," in *ADBIS*, 2007.
[17] H. Cao, O. Wolfson, and G. Trajcevski, "Spatio-temporal Data Reduction with Deterministic Error Bounds," in *DIALM-POMC*, 2003.
[18] N. Meratnia and R. By, "Spatiotemporal Compression Techniques for Moving Point Objects," in *EDBT*, 2004.
[19] Y. Tao and D. Papadias, "The mv3r-tree: A spatio-temporal access method for times- tamp and interval queries," in *VLDB*, 2001.
[20] M. Hadjieleftheriou, G. Kollios, J. Tsotras, and D. Gunopulos, "Indexing Spatiotemporal Archives," *International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 143–164, 2006.
[21] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer, "Towards an Analysis of Range Query Performance in Spatial Data Structures," in *PODS*, 1993.
[22] J. V. den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger, "XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries," in *VLDB*, 2001.