

# StarTrack Next Generation: A Scalable Infrastructure for Track-Based Applications

Maya Haridasan, Iqbal Mohomed, Doug Terry, Chandramohan A. Thekkath, and Li Zhang

*Microsoft Research Silicon Valley*

## Abstract

StarTrack was the first service designed to manage tracks of GPS location coordinates obtained from mobile devices and to facilitate the construction of track-based applications. Our early attempts to build practical applications on StarTrack revealed substantial efficiency and scalability problems, including frequent client-server roundtrips, unnecessary data transfers, costly similarity comparisons involving thousands of tracks, and poor fault-tolerance. To remedy these limitations, we revised the overall system architecture, API, and implementation. The API was extended to operate on collections of tracks rather than individual tracks, delay query execution, and permit caching of query results. New data structures, namely track trees, were introduced to speed the common operation of searching for similar tracks. Map matching algorithms were adopted to convert each track into a more compact and canonical sequence of road segments. And the underlying track database was partitioned and replicated among multiple servers. Altogether, these changes not only simplified the construction of track-based applications, which we confirmed by building applications using our new API, but also resulted in considerable performance gains. Measurements of similarity queries, for example, show two to three orders of magnitude improvement in query times.

## 1 Introduction

The easy availability of function-rich mobile devices has fueled significant interest in the “mobile internet”, where mobile devices access internet-based services and web applications. Mobile devices that can determine their own physical location are adding to this trend by facilitating the development of diverse location-based services. In addition to individual coordinates, “tracks” — time-ordered sequences of GPS locations recorded by mobile devices — enable many location-oriented applications,

varying from personal applications such as trip planning and health monitoring, to social applications such as ride-sharing and urban sensing.

StarTrack, introduced in an earlier paper, was the first service designed to manage tracks from mobile devices and to facilitate the construction of track-based applications [3]. That paper was primarily focussed on identifying a rich class of interesting personal and social applications that exploited histories of tracks; not much attention was paid to implementing the service at scale or building applications. Indeed, the entire implementation relied heavily on the services of a single database server with a thin software veneer providing an API. No applications were built using this API. Our first attempt to build realistic applications using this system revealed many shortcomings: principally inadequate performance, scalability, and fault-tolerance. Some of these, e.g. fault-tolerance, arose out of inadequate system structure in the original implementation. But by far most of the shortcomings arose out of a mismatch between the API provided by the system and what was required by applications. Specifically, several functions that were necessary for applications were either missing in the API or needed to be synthesized from lower-level primitives of the API. This mismatch led to costly and unnecessary client-server communication and data transfer. In addition to these deficiencies, our original system implemented common operations inefficiently (e.g. track comparisons).

This paper describes how the design and implementation of StarTrack have evolved non-trivially to address real-world issues of dealing with tracks. Our experience with track-based applications is admittedly limited. We do not claim our API is universal or fundamental in any sense; it will undoubtedly evolve as we encounter new classes of applications that we have not anticipated. Nonetheless, we believe our work and experience to date will be beneficial to researchers and practitioners in this rapidly growing field.

In general, we found managing and providing semantically rich operations on tracks to be surprisingly difficult. Track queries are complex because they involve geographic and similarity constraints, and a naive solution requiring expensive evaluation of these constraints does not scale to real-world online demand.

The main insight we use in tackling the complexity of tracks is to recognize that tracks tend to be repetitive. Repetitiveness arises from two distinct sources. An individual tends to follow substantially similar routes in his day-to-day life. This intuition is supported scientifically by a recent study in Science [23]. Second, the vast majority of tracks are collected on roads and highways, again leading to significant overlap in tracks even if they are from different users.

This insight permeates all parts of our revamped StarTrack infrastructure. We made several changes to our system. In some cases, we needed new techniques and data structures; in other cases, we used more established techniques, but synthesized in novel ways, to support a new class of track-based applications efficiently.

The changes to our system fall into four broad areas:

**API Changes.** All operations in our original API dealt with individual tracks, often causing entire sets of tracks to be moved repeatedly between the service and applications. StarTrack currently supports a “track collection”, representing a set of tracks. Several functions in the API now operate on and return results as track collections. This change had several benefits. Apart from the obvious ease of programming, it afforded StarTrack opportunities to optimize the performance of specific operations through delayed and partial evaluation of these collections. Caching of both full and partial results also became possible.

**Changes in Track Representation.** We quickly discovered dealing with “raw” tracks by themselves to be inefficient. We now use a “canonical” representation for tracks, where tracks are represented as a sequence of points drawn from a fixed set, such as road intersections. Canonicalization benefits many aspects of the system. It reduces the computational costs of track comparison while improving its accuracy. As a consequence of improved accuracy, we are able to group a user’s similar tracks more effectively and maintain a small set of representative tracks that captures the essentials of a large set of tracks. Many applications only need to operate on the set of representative tracks, leading to significantly fewer operations, better caching of data, and consequently, better performance.

**Changes to On-Disk and In-Memory Data Structures.** The original StarTrack API was implemented as a thin veneer on top of a geospatial database system. While simplifying the implementation, this resulted in

poor performance for many operators. The changes in the API and canonicalization described above allowed us to build specialized in-memory data structures to augment the database tables. Operations that had low performance are now optimized by using in-memory quad-trees or a novel structure called a *track tree* described in Section 3.3. In addition to these in-memory data structures, we reorganized the database layout to include a table of representative tracks for each user (as mentioned above) and other tables that aid in handling operations with geographical constraints.

**Structural Changes.** Our original prototype consisted of a single server process that stored tracks in a centralized database and implemented an API to access these tracks. This single server implementation clearly did not scale to a large number of tracks or provide fault-tolerance. In the new system, a set of StarTrack server machines connects to another set of database servers. Applications use a StarTrack clerk, which implements the API and makes remote procedure calls (RPCs) to the StarTrack servers as necessary. It also deals with retrying requests on server failures, and balances RPC requests amongst servers.

We detail our changes further in the rest of the paper (Sections 2–4), describe two scalable, robust, and efficient applications they enabled us to build (Section 5) and summarize their performance impact (Section 6).

## 2 Application Programming Interface

The interface exported by the StarTrack service has undergone multiple revisions based on our experience building realistic applications. This section describes the key elements of the new application programming interface; space restrictions prevent us from describing the complete API.

### 2.1 Track Collections

The new StarTrack interface supports the notion of a *track collection*, an abstract grouping of tracks, where the application supplies the criteria for grouping. Track collections can, in turn, participate in other StarTrack operations. All non-trivial operations in the StarTrack API take a track collection as an argument.

Track collections have two significant advantages:

**Implementation Efficiency.** They allow the server to treat the set of tracks that are repeatedly accessed together as a single entity for the purposes of caching. They also allow the server to construct specialized data structures that operate exclusively on these tracks, making these operations more efficient. Furthermore, by having applications and the service refer to a potentially

large collection of track identifiers by a single identifier, we reduce the communication costs of transmitting the identities of individual tracks between them.

**Programming Convenience.** Applications often want to constrain operations to tracks that belong to a particular community or cohort. For example, a social application might wish to operate on the tracks of a user and his group of friends. Track collections allow such an application to create an aggregation of the tracks in which it is interested and enable it to operate on such groups more conveniently.

Track collections are created by using the *MakeCollection* procedure (see API Fragment 2.1). *MakeCollection* takes as its first argument a set of criteria to select a group of tracks from all tracks in the system. Individual criteria can be composed out of three elements: *geographic, time, user*. The first two elements have fairly simple semantics: a geographic element is specified by a physical geographical region and a time element is specified by a time interval. The user element consists of two subfields: a unique identifier that specifies the user and a string field that specifies an XPATH query. The query is applied to the user metadata that is stored in the track by the application.

```
TrackCollxn MakeCollection(GrpCriteria[] gCrit,  
                           bool unique);
```

**API fragment 2.1:** Operation to create a track collection.

The second argument is a boolean that indicates whether the system should return only “unique” tracks. Two canonical tracks are considered unique if their starting points (as well as ending points) are “close” to each other, and their paths are highly “similar” to each other. Similarity is more precisely defined below when we discuss the *GetSimilarTracks* function. Parameters that decide if the start/end points are “close” to one another and if tracks are highly similar are defined by the infrastructure. These are described further in Section 4.1.

We provide applications the option to specify the unique flag for two reasons. People tend to travel the same routes habitually, leading to multiple highly similar tracks that only differ in time. Meanwhile, many applications are only interested in distinct routes without requiring knowledge of the precise times at which the route was traveled. These applications greatly benefit from using *MakeCollection* with the `unique` flag set to true since it significantly reduces the number of tracks in the returned collection. If instead an application needs per track information, for instance, if it needs to know how fast the user travels on a particular road segment, setting `unique` to

false will retrieve all the relevant tracks with detailed information.

Two simple code segments calling *MakeCollection* are shown in Examples 2.1 and 2.2. The first example collects the tracks of user Uriah between 8AM and 10AM. The second shows how metadata information is used to create a track collection of all employees of an organization.

---

**Example 2.1** Uriah’s tracks between 8AM and 10AM.

---

```
GrpCriteria[] gCrit = new GrpCriteria[2];  
UserCriteria uc = new UserCriteria();  
uc.Username = "Uriah";  
TimeCriteria tc = new TimeCriteria();  
tc.StartHour = 8; tc.EndHour = 10;  
gCrit[0] = uc; gCrit[1] = tc;  
TrackCollxn tcUriah;  
tcUriah = MakeCollection(gCrit, false);
```

---

**Example 2.2** Tracks of all employees of the Wickfield corporation. The metadata string is an XPATH query, shown here in simplified syntax for formatting reasons.

---

```
GrpCriteria[] gCrit = new GrpCriteria[1];  
UserCriteria uc = new UserCriteria();  
uc.metadata = ``Employer = Wickfield``;  
gCrit[0] = uc;  
TrackCollxn tcWField;  
tcWField = MakeCollection(gCrit, true);
```

---

## 2.2 Manipulating Tracks

Tracks can be manipulated in several ways; we describe a few representative operations. We have chosen these because they embody the most significant changes we made to the original prototype. Other operations are essentially unchanged from our previous API.

*JoinTrkCollections* takes two or more track collections and creates a new track collection that is the union of all the constituent tracks. The second argument allows the resulting track collection to retain only unique tracks. *SortTracks* takes a track collection and orders the constituent tracks in the collection according to one of a set of predefined attributes. Examples of attributes we have implemented are `LENGTH` and `FREQ`, which refer to the length of the track and its frequency of occurrence within that track collection.

Many track-based applications need to determine whether tracks are similar to one another. Given two tracks, we define track similarity as the ratio of the length of all the segments that are common to both of them divided by the length of the union of all segments present in either of them (Figure 1(a)). *GetSimilarTracks* is given

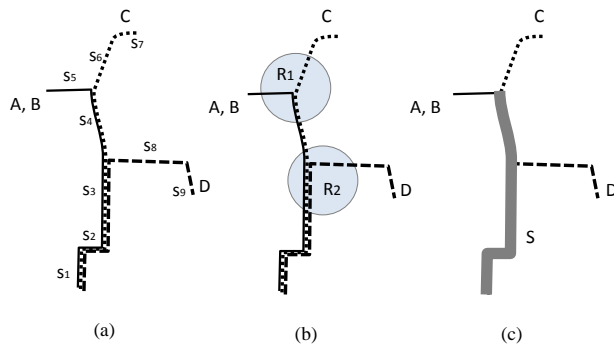


Figure 1: (a) the similarity between tracks A and B is 1 and between A and D is  $(l_1 + l_2 + l_3)/(l_1 + l_2 + l_3 + l_4 + l_5 + l_8 + l_9)$ , where  $l_i$  is the length of segment  $s_i$ ; (b) A,B,C are the tracks that pass by the areas  $R_1$  and  $R_2$ ; (c) S is the common segment of A,B,C,D with frequency threshold set to 0.6.

a track collection and a reference track and selects from within the collection all tracks that are similar to the reference track. The returned track collection is sorted by similarity. The degree of similarity is controlled by the third parameter.

Track-based applications can find tracks that pass within close proximity of a location by calling *GetPassByTracks*. *GetPassByTracks* is given a track collection and an array of *Area* objects and returns all tracks in the collection that pass through all the areas (Figure 1(b)).

*GetCommonSegments* takes a track collection and a frequency threshold and returns the road segments shared by at least that fraction of the tracks in the collection. These road segments are merged into the smallest number of contiguous routes possible (see Figure 1(c)). This operation is useful for the application to retrieve a succinct summary of a potentially large set of tracks.

Tracks within a *TrackCollxn* object can be retrieved via the following two functions (See API Fragment 2.3). *GetTrackCount* returns the number of tracks in a track collection, and *GetTracks* returns *count* tracks beginning at the *start* location within a track collection.

### 3 StarTrack Server Design

This section describes three changes to the StarTrack server design that we consider most significant.

#### 3.1 Canonicalization of Tracks

In our first implementation, we stored users' latitude and longitude coordinates directly in the system. While this design choice was intuitive and useful in some circumstances, it was problematic in many others. Recall that

```
TrackCollxn JoinTrkCollections(TrkCollxn tCs[],
                               bool unique);

TrackCollxn SortTracks(TrkCollxn tC,
                       SortAttribute attr);

TrackCollxn GetSimilarTracks(TrkCollxn tC,
                              Track refTrk, float simThresh);

TrackCollxn GetPassByTracks(TrkCollxn tC,
                             Area[] areas);

TrackCollxn GetCommonSegments(TrkCollxn tC,
                               float freqThresh);
```

**API fragment 2.2:** Operations to manipulate a track collection.

```
int GetTrackCount(TrkCollxn tC);
Track[] GetTracks(TrkCollxn tC, int start,
                 int count);
```

**API fragment 2.3:** Retrieval operations on a track collection.

coordinates are samples of a path taken by a user. The same path taken by different users may be sampled at different points. Also, sampling is inherently error-prone due to limitations in current localization techniques [8]. For these reasons, two identical paths can lead to widely different sampled coordinates, making it difficult to classify them as equal. In the new system, we “canonicalize” paths to eliminate spurious variability in the sampled coordinates. In this context, canonicalization means that we convert a path to another path that only passes through a set of “standard” points drawn from a (large) fixed set. We refer to the portion of the path between two such points as a segment.

There are several methods to canonicalize tracks. One intuitive way is to overlay a fixed grid on the geographic region and to map each coordinate to a grid intersection point. A variation on this technique is to pick a suitably weighted interior point within the grid instead of a corner.

A fundamental shortcoming of approaches based on a fixed grid is that the grid is artificially created and does not adapt to users' tracks. Grids may be too fine-grained, in which case canonicalization provides no benefits, or too coarse-grained, in which case important features of tracks are lost.

Instead of using an artificial grid, we can often use the more natural and adaptive grid imposed by streets and highways. Canonicalizing based on street maps is called *map matching* and is desirable in cases where roadmaps of the region exist. A track after canonicalization is mapped to a path in the roadmap. A path consists of one or more street segments and is stored as a sequence

of the endpoints of the segment(s). StarTrack uses a map matching approach using hidden Markov models designed by Krumm *et al.* [17, 20].

The performance of canonicalization is dependent on three factors: the sampling rate of a track (i.e., the number of GPS points in the track), the length of the track, and the amount of GPS noise introduced into the samples. In our system, canonicalization is done offline as a pre-processing step. Since the performance of canonicalization is not that critical in our system, we do not present detailed results. With some performance tuning, StarTrack can canonicalize a track with average trip length of about 20 km and 400 GPS samples in under 250 ms.

Canonicalization has two key advantages that translate into performance savings. First, StarTrack can compare two segments for equality without using expensive geographic constraints. Equality of segments is used within the inner loop of the procedure that finds similar tracks, which in turn is a very common operation in applications. Second, canonicalization tends to create larger numbers of identical segments. This often allows us to access and manipulate a single representative segment rather than dealing with individual segments. It also allows StarTrack to identify duplicate tracks more accurately and reduces the number of tracks it needs to process for various operations.

Canonicalization based on road networks is appropriate for regions that have a mature road network and a stable map. When road networks are not available, we may utilize technologies for constructing road maps from user tracks [5, 7].

### 3.2 Delayed Evaluation

We found that applications typically make several API calls to narrow down the set of tracks they want to retrieve. Our implementation of the API therefore delays the evaluation of the tracks in a track collection until one of the two retrieval functions in API Fragment 2.3 is called. This technique saves multiple roundtrips between the StarTrack clerk and servers. Furthermore, it allows the StarTrack server flexibility in the queries it issues to the database and in the choice of data structures it builds for different retrieval operations.

When a client invokes a *MakeCollection* operation, the client-side stub marshals an efficient description of the call arguments and a small integer representing the procedure name. We call the resulting structure a *descriptor*. The stub sends the descriptor to the server, which stamps it with the current time to capture the database contents at that instant and returns it.<sup>†</sup> We require that the timestamp be in the past with respect to the time on the database

<sup>†</sup>There are well-known ways to avoid this RPC call, but we have chosen not to implement them for simplicity.

server. Assuming that tracks are not deleted from the system, this guarantees that multiple evaluations of a track collection will always return the same set of tracks.

Operations such as *JoinTrkCollections*, *GetPopularTracks*, *GetSimilarTracks*, and *GetPassByTracks* create compositions of these descriptors (at the client stub) with no communication to the server and no additional timestamps. We refer to these compositions as *compound descriptors*. These are organized as a tree, with the leaves being a simple timestamped descriptor.

Notice that all descriptors (compound or otherwise) contain information about the invoked function and the arguments, which together can be used to construct a track collection. In this sense they can be viewed as a closure [18] or as a specialized form of a logical view from the database literature [9].

Our use of timestamped descriptors is a tradeoff between efficiency and freshness. Timestamps imply that the application sees data as it existed in the database at a particular point in time, not necessarily the latest data. It allows the StarTrack server to cache the contents of the database in an in-memory data structure, or discard it at will and reevaluate it later, while providing easy to understand and consistent semantics to the application. It also allows a client to present the descriptor to a different StarTrack server if needed for load-balancing reasons or if the original server crashes. Re-evaluating a descriptor is guaranteed to yield the same result anywhere in the system because the operations are deterministic, and the timestamp acts as a snapshot of the database (provided that tracks are not deleted from the system). If freshness is more important for an application, it can recreate the track collection as often as needed.

The evaluation of a descriptor yields different types of in-memory data structures. For example, the evaluation of a descriptor constructed by *GetSimilarTracks* may (but need not) create a data structure called a track tree. A descriptor created by *GetPassByTracks* can result in a quad-tree [10]. The results of evaluating other descriptors are typically stored as a simple set of tracks.

### 3.3 Track Tree

In our experience, when two tracks overlap, they usually do so on one or very few contiguous segments. We exploit this property to build a hierarchical data structure called a *track tree*, which is used to speed up the retrieval of similar tracks.

Each road segment is represented as a leaf node in a track tree. For each leaf node, the track tree records all tracks that contain that particular segment. Once all the segments in a track collection are stored as leaf nodes, pairs of nodes that refer to geographically adjacent segments are considered for merging to form interior nodes

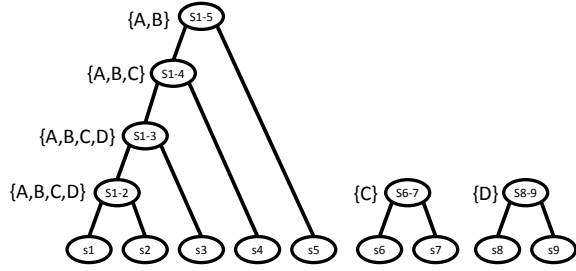


Figure 2: The track tree of the set of four tracks shown in Figure 1(a). Each node, except for leaf nodes, is annotated with the set of tracks that contain it.

of the tree. Whenever there is choice of pairs of nodes to merge, the pair that has the highest number of tracks in common is picked. This process is continued iteratively up the tree. When merging two nodes, all tracks belonging to both children nodes are included in the parent node as well. By this construction, each node in the track tree represents a contiguous sequence of road segments. In addition, the segment is more likely to be shared by multiple tracks.

Figure 2 shows the track tree for the sample four tracks in Figure 1(a). As shown in Figure 1(a), tracks A and B are identical and consist of segments S1, S2, S3, S4, and S5. Tracks C and D share common segments with A and B. Segments shared by larger numbers of tracks are favored when merging nodes, which explains why segments S1 to S3 are merged together, instead of other combinations, such as S2 to S4. Using this tree, tracks A and B can be described by one single node (S1-5), and tracks C and D can be described by two nodes each: Track C by S1-4 and S6-7 and Track D by S1-3 and S8-9.

Track trees are used to accelerate several API operations. In *GetCommonSegments*, after we identify the road segments shared by sufficiently many tracks, as indicated by the given threshold, we use a track tree to organize them into a small number of contiguous tracks. This is done by merging up in the tree those nodes corresponding to these road segments. Given the way a track tree is constructed, this usually results in a small number of nodes, corresponding to a small number of contiguous tracks.

Another API operation enabled by a track tree is *GetSimilarTracks*. Implementing this function as a database operation is inefficient because there is little match between our similarity semantics and the primitives supported by spatial databases.

With a track tree, StarTrack can quickly find a set of tracks with a given degree of similarity to a specific track T (See Code Segment 3.1). First, StarTrack identifies the set of all nodes (interior and leaf) covered by T. In order to do this, T is initially broken into smaller seg-

ments. StarTrack then identifies the leaf nodes in the track tree that correspond to these segments. Next, it identifies pairs of adjacent nodes that have a common parent node, includes the parent into the set, and iterates until no such parent exists. These steps are encapsulated in the function `Map`.

The *GetSimilarTracks* operator then sorts the nodes in T by decreasing order of length. It sequentially scans each node, examining the set of tracks containing it, and outputs tracks that are at least `simThresh` similar to the query track. This process stops when it has found sufficiently many tracks as defined by the `maxCount` parameter, or when it has examined sufficiently many tracks. Recall that the client supplies the `simThresh` parameter (as part of the *GetSimilarTracks* call), as well as the `maxCount` parameter (as part of the *GetTracks* invocation, which triggers the evaluation of the descriptor). This process will not produce any false positives (i.e., tracks that purport to be similar but are not), but it could miss some highly similar tracks. The percentage of such misses is quite small when the similarity threshold is reasonably high, as our experimental results show (see Figure 7(c) in Section 6).

---

**Code segment 3.1** Pseudo-code for implementing *GetSimilarTracks* using *tracktree*.

---

```
Track[] GetSimilarTracks(TrackTree trackTree,
    Track T, double simThresh, int maxCount)
{
    TrackTreeNode[] nodes = trackTree.Map(T);
    SortByDescLength(nodes);

    SortedList<Track> results; int examined = 0;
    foreach(node in nodes) {
        foreach(candidate in node.tracks) {
            if(T.Similarity(candidate) >= simThresh)
                results.Add(candidate);
            examined++;
            if((results.Count >= maxCount) ||
                (examined >= 6 * maxCount))
                return results;
        }
    }
    return results;
}
```

---

Similar to other in-memory data structures in StarTrack, a track tree is cached in memory until evicted under the caching policy: LRU in our implementation. Since track collections are immutable, we do not update data structures during their life time. However, the track tree structure allows for efficient insertion of new tracks, and whenever a track collection is created by building upon an existing track collection, an existing underlying track tree may be copied and updated.

## 4 Storage Platform Design

As previously described, we build and maintain in-memory data-structures at the StarTrack servers, and use a different set of database servers to store data persistently. StarTrack always checks if tracks can be found in the in-memory data-structures before fetching them from the database.

StarTrack uses Microsoft's SQL Server 2008, which supports the notion of geospatial objects as a fundamental data type. Data is partitioned across multiple machines, and partitions are replicated using chained declustering [12], which provides the necessary scaling properties as well as automatic dynamic load-balancing and fault-tolerance.

### 4.1 Database Tables

The principal on-disk data structure consists of 5 tables stored in SQL Server.

**User Table.** This consists of a set of records for each user containing a unique system-assigned user identifier and other personal information.

**Track Table.** Every track is assigned a unique identifier, consists of a set of time-stamped latitude and longitude coordinates, and is stored in a single row in the table. Both the raw and the canonical versions of tracks are stored in the same table.

**Representative Track Table.** This table maintains a set of representative tracks per user and allows StarTrack to often avoid searching the larger Track Table. Each record stores information related to a single representative track: the canonical coordinates, the owner, and a count of how many actual instances of this representative track exist in the Track Table. Upon insertion of a track into the Track Table, StarTrack checks if there exists a representative track that matches the new track. If so, the new track is not inserted into the Representative Track Table, but the count of the matching representative track is incremented. The count serves as indication of the popularity of a given representative track and is used by StarTrack operations for ranking purposes.

Two tracks are considered as matching if their start points are within 100 m of each other, if their end points are within 100 m of each other, and if the tracks are at least 90% similar. The choice of these parameters is fixed by the infrastructure and cannot be changed by individual applications. It is based on expected errors in GPS measurements, as well as cost/benefit tradeoffs, and is not as Procrustean as one might imagine. The values chosen determine the size of the Representative Track table — high start/end point buffer values and low track similarity values result in a smaller table of unique tracks, but

applications may lose the ability to discriminate between tracks. The size of the table, in turn, affects the speed of many functions in the API that must access that table.

**Coordinate Table.** During the map matching process, the set of coordinates in a path is drawn from a finite list of points, which depends on the particulars of the map data used for canonicalization of tracks. Each record in this table maps a location identifier to a pair of coordinates. This particular table is immutable, replicated on each database server, and not partitioned.

**Coordinate to Track Table.** This table maps coordinates to tracks that go through them. We use it to speed up the location of tracks that pass through certain geographic boundaries.

StarTrack allows three types of criteria in fetching tracks from the database: user, time, and geographic region. Region-based queries may be performed by leveraging the geospatial functions provided by modern database systems, which support specialized indexing schemes. Such systems must be used with care because costs are still significant when indexing large numbers of complex geospatial objects such as tracks.

In the original StarTrack implementation, we used the geospatial primitives of the database to treat each track as a separate object and created a geospatial index over all such objects. Now, we maintain a geospatial index on the Coordinate Table alone, thereby reducing the number of objects on which the geospatial index is maintained. We use this index to find all locations that match a given geographic query. We then use the Coordinate To Track Table to look up all tracks that go through these locations. This is feasible precisely because of the canonicalization pre-processing step.

The Coordinate Table and its geospatial index are maintained by the database server and portions of them may be cached in memory. We present a comparison of the original and new approaches in Section 6.2 (Figure 3). If necessary we can further speed up our design by not storing the Coordinate Table in the database server and can instead store it in memory and index it using an in-memory quad-tree.

### 4.2 Database Server Organization

The tables mentioned above are partitioned across multiple database servers. Based on StarTrack's search criteria options, we considered two partitioning schemes: by geography and by user identifier.

We decided not to partition by geography, since over time it would lead to increasing numbers of tracks that span geographic regions, therefore having to span servers.

We opted for partitioning data by user identifier, keeping all data referring to a single user in a single database server. This organization allows user-constrained queries to be sent to a single database server, while requiring geographic queries to be sent to all database servers.

Data is mirrored in the system. Each database server acts as the primary for one partition of each table, and as the mirror (or secondary) for its neighbors' partitions. A primary database server processes read and write requests from clients, while a mirror server only handles read requests.

StarTrack servers are clients of the database servers, and evenly distribute reads amongst the replicas. When a database server fails, the server that mirrors the partitions on the failed server takes over as primary for the partitions. The StarTrack servers direct write traffic to the new servers and in addition, distribute the read requests uniformly among all the replicas using chained declustering, as described by others [12, 19].

## 5 Applications

We explored scenarios where a single user's data can be used to personalize her experience based on her habitual tracks, for applications such as personalized advertising, recommendation systems, and health monitoring. On the other end of the spectrum, social applications, where the set of tracks from a group of friends or even a broader community are used, may help provide enhanced services to users. Examples include those related to urban sensing, collaboration, discovery of new areas, and shared experiences.

To illustrate the usefulness and evaluate the performance of StarTrack services, we describe two of the applications we built.

While both applications were non-trivial to write, the use of our API significantly simplified their construction. In fact, the application logic in both examples is succinctly captured in a few code snippets. Our general experience is that StarTrack provides an intuitive, flexible, and efficient way to program track-based applications.

### 5.1 Ride-Sharing Service

Ride-sharing has long held the promise of reducing energy consumption. Transit departments in many major metropolitan areas now offer on-line ride-sharing services or portals (see for example, King County Metro Ride [15]). One challenge in building an effective ride-sharing service is to discover ride-share partners who travel on similar routes.

With StarTrack, these ride-matching services are easily built. The service can build a `TrackCollection` for the employees of the same company or for a person's

social network, or for a group of people who have subscribed to a transit service. Code Segment 5.1 constructs a track collection for a community of users. Code Segment 5.2 identifies potential ride-sharing partners based on the similarity of their travel patterns.

---

**Code segment 5.1** Set up a community's regularly traversed tracks where the community is defined through supplied `SearchCriteria`.

---

```
TrackCollxn getCommunityTracks(SearchCriteria sc,
                                int count)
{
    TrackCollxn tc = MakeCollection(sc, true);
    return Take(SortTracks(tc, FREQ), count);
}
```

---

---

**Code segment 5.2** Find ride-share candidates with similar travel patterns. `findOwners` is a client-side function that takes a set of tracks and returns the list of users who own them.

---

```
List getRideShareCandidates
(TrackCollxn communityTC, string username)
{
    UserCriteria uc = new UserCriteria();
    uc.Username = username;
    TrackCollxn userTC =
        MakeCollection(uc, true);
    Track[] popularTracks =
        GetTracks(SortTracks(userTC, FREQ),
                  0, 10);
    List<TrackCollxn> similarTC;
    foreach(Track track in popularTracks) {
        TrackCollxn tc = GetSimilarTracks(
            communityTC, track, 0.7);
        similarTC.Add(tc);
    }
    Track[] similarTracks =
        GetTracks(JoinTrackCollections(similarTC),
                  0, 100);
    return findOwners(similarTracks);
}
```

---

Another usage scenario is when a user needs a ride between two specific locations. This can be done easily by calling `GetPassbyTracks`.

It is important to note that the ride-sharing service based on StarTrack offers more flexibility than conventional services. For instance, since a rider's entire route is known, rather than just his start and destination, it allows the service more latitude in arranging pick-ups and drop-offs along the route.

### 5.2 Personalized Driving Directions

Current navigation systems and online map services provide detailed turn-by-turn driving directions. Because



StarTrack knows what routes a person has taken in the past, as well as how recently and how frequently, an application could easily use StarTrack to provide personalized driving directions.

For example, instead of providing detailed turn-by-turn instructions on how to get to the freeway from the person’s house, the directions might simply say “Get on Highway 101 heading south” and then provide detailed directions from that point.

---

**Code segment 5.3** Construct a user’s familiar segments.

```
TrackCollxn getFamiliarSegments(string username)
{
    UserCriteria uc = new UserCriteria();
    uc.Username = username;
    TrackCollxn uTC = MakeCollection(uc, true);
    // Pick the 10 most frequently occurring
    // tracks.
    TrackCollxn pplrTC =
        Take(SortTracks(uTC, FREQ), 10);
    TrackCollxn familiarTC =
        GetCommonSegments(pplrTC, 0.2);
    return familiarTC;
}
```

---

The application we built uses the Bing Map service and the StarTrack infrastructure. A user inputs start and destination locations, and the application uses Bing to get turn-by-turn directions for that route. Next, the application uses StarTrack to obtain the set of “familiar segments” for that user, as shown in Code Segment 5.3.

Having obtained the familiar segments for the user, the application identifies portions of the route returned by Bing that overlap with the familiar segments and uses the result to prepare personalized driving directions (we omit further description of these steps given that they are performed locally by the application and do not involve calls to StarTrack).

## 6 Evaluation

This section evaluates the performance of the StarTrack service. To study the system at scale, we used synthetically generated tracks. We also ran experiments with actual tracks collected by users of GPS-equipped mobile devices, but omit the results since they are similar to those performed with synthetic tracks, and given that we only have a limited number of real tracks.

We focus on the costs of executing track operations that involve (a) geographic constraints and (b) comparisons of tracks. These operations are the most difficult to build efficiently, and are also among the most commonly occurring in the track-based applications that we built. We also report on the performance of two applications.

Our experiments were all conducted on 2.6 GHz AMD

Opteron quad-core processors with 16 GB memory, running Windows Server 2003.

### 6.1 Synthetic Tracks

We generated synthetic tracks based on the salient features observed in a dataset of approximately 16,000 real tracks followed by 252 users over 2-week periods in Seattle, WA [16]. In our model, each person has fixed locations for home and workplace, and a number of “errand” locations that represent places they go less frequently. On weekdays, a person travels between the assigned home and work locations during the common morning and evening commute hours. Sporadically on weekdays and more often on weekends, a person carries out a number of errands.

After choosing the start and end locations for each trip, we calculate the shortest path as well as its duration between these points on a graph of road networks. We then sample and perturb each path to simulate noise in the sampling and localization of the data and treat the resulting points as a track.

Our early experiments indicated that some features of tracks have a pronounced effect on performance while others do not. Specifically, performance is affected by the following:

- Number of tracks. The larger the number of tracks, the greater the computational and storage overhead.
- Length of tracks. The number of points in a track has an impact on performance. Assuming tracks are canonicalized, the number of points is proportional to the length of the tracks.
- Covered region. The region over which the tracks are generated has an impact on track density (i.e., number of tracks that pass through a unit area). As track density increases, the computational burden imposed on our algorithms increases. For example, the same geographic query returns more results and therefore incurs more computational cost when the density of tracks is higher.

We devised our model to allow us to control these key features. Our belief is that, at least for the purpose of performance evaluation, any model that allowed these features of tracks to be varied would be adequate.

For our scalability experiments, we generated synthetic tracks for a 3-month period and 18,000 users in Santa Clara County. This resulted in a total of over 4.5 million tracks. On average, each track is 20 km long and contains 400 GPS samples that yield on average 163 points after canonicalization.

## 6.2 Performance of Geographic Queries

One of StarTrack’s most important operations is querying based on geographic constraints. Some of these operations require a round-trip to the database server, while others can be optimized by an in-memory cache. In our API, geographic queries show up in two forms. First, in *MakeCollection* an application can specify a geographic region constraint. Second, *GetPassByTracks* allows an application to select those tracks in a track collection that pass within specific areas. The first query involves retrieving tracks from a database, while the second involves retrieving tracks from a pre-computed track collection, which can be sped up in memory.

**Geographic queries to the database.** Although we do not focus on studying the performance of the spatial features of the database, we investigate how best to use them to improve simple geographic queries used to pre-filter tracks brought into memory.

We compared two ways to store tracks and construct the necessary indices. In the first approach, used in our original prototype, we treat each track as a separate geospatial object and create a spatial index over all tracks. This index is used to retrieve all the tracks intersecting the query region. The second approach, used by StarTrack, involves the use of two additional tables, the Coordinate Table and the Coordinate to Track Table, as described in Section 4.1. In this approach, a spatial index is built only on the Coordinate Table.

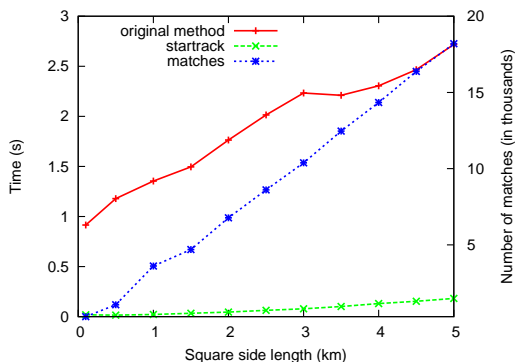


Figure 3: Query time with and without the Coordinate Table when searching for tracks that intersect square regions of increasing side lengths. Secondary  $y$ -axis shows average number of tracks matched.

Figure 3 presents the query time for both approaches when we vary the area of the query region on a set of 100,000 tracks. It also shows the average number of matched tracks on the secondary  $y$ -axis. Isolating the need to execute geographic queries to a small set of distinct points through the use of the Coordinate Table leads

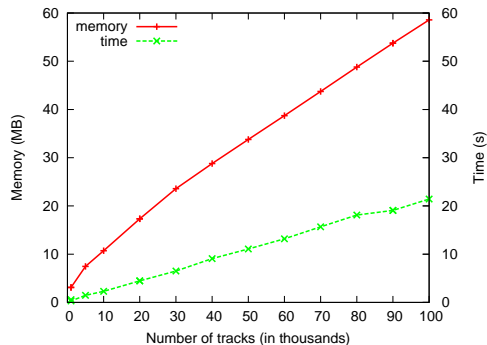


Figure 4: Memory usage and construction time of the quad-tree for different sizes of tracks.

to significant performance benefits. This enhancement is only possible due to track canonicalization.

**Geographic queries to in-memory data structure.** Recall from Section 3.2 that the evaluation of a *GetPassByTracks* operation triggers the construction of an in-memory quad-tree, in the expectation that the data will be repeatedly accessed in the future. Canonicalization tends to lower the number of unique coordinates in tracks, speeding up the construction time for quad-trees, as well as the execution time of subsequent requests against it. Figure 4 shows the cost of constructing a quad-tree. Building the quad-tree itself requires little space and time since the number of unique coordinates is small and levels off when the tracks cover a large region. Both the memory and time needed are linear in the number of tracks, and are mostly spent on building an index from coordinates to their containing tracks.

Figure 5 presents the time to query a quad-tree with varying numbers of tracks and region sizes. In all cases, the query time is very low. For example, it takes about 1 ms for a region with a 5 km radius on 100,000 tracks. The query time is fairly insensitive to the number of tracks because the structure of the quad-tree is determined by the unique coordinates. On the other hand, the size of the query region affects the times since it determines the number of quad-tree cells to be visited.

## 6.3 Performance of Track Comparisons

A common query in track based applications is to retrieve tracks based on similarity. Typically, an application has a track collection and a “query” track and needs to find tracks in the set that are most similar to the query track.

We compare the performance of our technique using a track tree to three alternative methods for ranking tracks based on similarity: (1) *Bruteforce*: The brute-force method compares the query track against every track in the collection and returns those with similarity

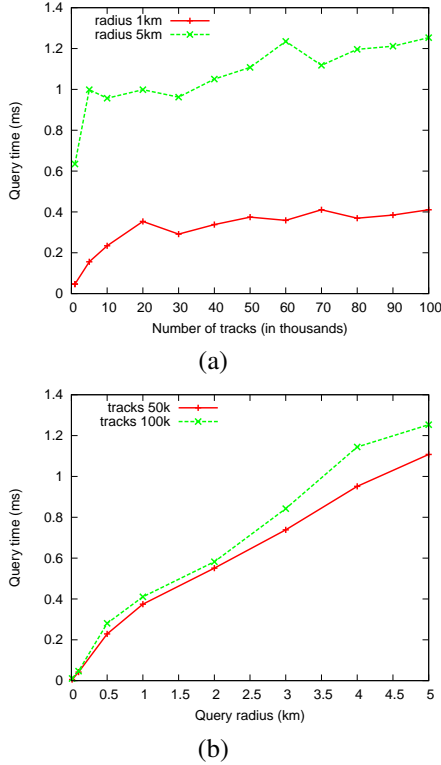


Figure 5: Time to query a quad-tree. (a) Query time for different numbers of tracks when size of the region is fixed to 1 and 5 km, respectively. (b) Query time for 50K and 100K tracks as the size of region is varied.

above a given threshold. For the bruteforce method, we assume all tracks are already in memory. (2) *In-memory filtering*: This method constructs an in-memory dictionary used to quickly look up tracks that contain any given point. For a given query track, we use this dictionary to identify all tracks that intersect it, after which we compute the similarity of each intersecting track to the query track, returning those above the threshold. (3) *Database filtering*: We store the set of tracks in the database, use a query to retrieve all tracks in the database that intersect the query track, and compute the similarity against the retrieved tracks.

We ran experiments with different numbers of tracks and queries with varying similarity thresholds.

Figure 6 shows the query time when using the various methods. The query time with the track tree method is dependent on the similarity threshold, unlike with the other three alternatives. In Figure 6, we present results for the track tree approach when the similarity threshold is 0.7 and 0.9. The experiments show that track trees lead to significantly more efficient queries when compared to the bruteforce method, achieving two to three orders of magnitude speedups. Although the in-memory filtering

method performs better than the bruteforce method, it is still significantly slower while consuming high amounts of resources for constructing and storing the in-memory dictionary. The database filtering method presented the worst performance.

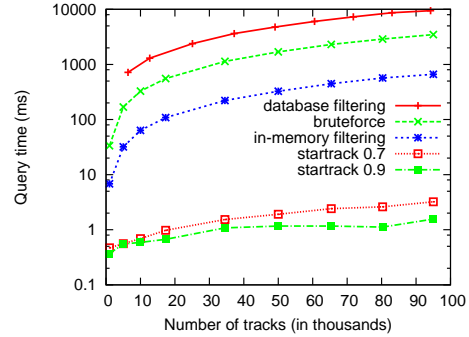


Figure 6: Query time comparison between StarTrack and three alternative methods. For StarTrack, results are shown for similarity thresholds of 0.7 and 0.9.

There is a cost associated with constructing a track tree that is at the heart of our technique. Figure 7(a) shows the memory usage and the time for constructing a track tree as a function of the number of tracks in the collection. Constructing a track tree takes linear space and slightly super-linear time as the height of the track tree grows logarithmically with the number of tracks. There is a tradeoff for using a track tree— it takes time to construct it, but once constructed, it leads to significantly optimized queries. From Figures 7(a) and 6, we calculate the “break-even” point, or the minimum number of queries such that the amortized query time using a track tree is lower than the query time of the bruteforce method. These break-even numbers are shown in Figure 7(b). As observed, the numbers grow slowly with the number of tracks, and are fairly small: below 80 for a track collection with up to 100,000 tracks.

One potential downside of the track tree approach is that while it is highly efficient at retrieving similar tracks and although it will never return tracks that do not satisfy the similarity threshold, it may not return all tracks above the given similarity threshold. Figure 7(c) shows the coverage of the track tree method. The graph shows the percentage of the expected tracks returned when using a track tree. We can see that the coverage increases for higher similarity thresholds. It returns over 90% of the tracks when similarity is above 0.7. We believe this is sufficient for typical applications, that are only interested in tracks with reasonably high similarity.

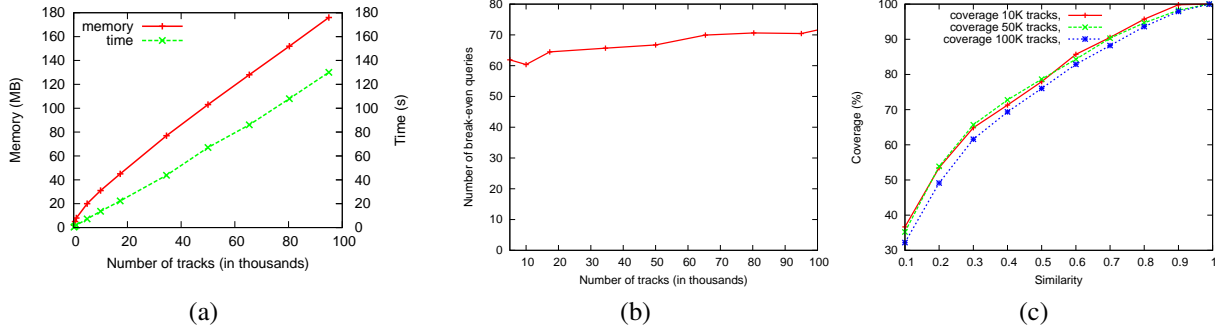


Figure 7: (a) Memory and processing time required for constructing a track tree. (b) Break-even number for use of a track tree. (c) Coverage of track tree approach as function of the similarity threshold (for 10K, 50K and 100K tracks).

## 6.4 Application Performance

We use the Ride-Sharing (RS) and Personalized Driving Directions (PDD) applications, presented in Section 5, to evaluate the overall performance of StarTrack. These two applications illustrate two different usage scenarios: RS creates a large track collection for repeated accesses while PDD creates many small per-user track collections.

We fixed the number of database servers to three and varied the number of StarTrack servers. To generate load on the servers, we ran multiple instances of these applications from a number of client machines.

### 6.4.1 Single StarTrack Server Experiments

The RS application identifies potential ride-sharing partners for a given user, and as presented in Code Segment 5.2, involves multiple calls to the StarTrack server. In our evaluation, we built a track collection with 50,000 unique tracks from which the application searches for similar tracks. We warmed up the server by constructing a track tree on the large set of tracks before sending it client requests. Figure 8(a) shows the response times for RS under varying request rates. Despite the more complex nature of the application, one StarTrack server is capable of satisfying 30 requests per second with a response rate of around 150 ms.

We ran experiments for the PDD application under two different types of load. In the first case, queries simulate users whose data has not been cached on the StarTrack server prior to the query. In the second case, we preload the cache with the in-memory data structures used to expedite the *GetCommonSegments* operation (*familiarTC* in Code Segment 5.3) invoked by the application.

Figures 8(b) and (c) plot the response times with varying request rates under the two types of loads. When the data is not cached, each server is capable of satisfying up to 30 requests per second without increasing the response time. The average response time prior to saturat-

ion is around 100 ms. The maximum server throughput increases to 270 requests per second and the response time falls to 60 ms when the data is previously cached on the server.

### 6.4.2 Scalability Experiments

For both applications, individual requests sent by the clients are entirely independent of one another. We tested StarTrack’s scalability by running the PDD application on multiple StarTrack servers. For this experiment we used the non-cached version of PDD, with the goal of exercising load on the database.

In Figure 9 we present the maximum throughput that the system is able to achieve with a varying number of StarTrack servers. As expected, the system scales linearly with the number of servers. Since PDD only retrieves a small number of tracks for each user, this experiment did not saturate the database servers.

From these experiments, we estimate the resources needed to satisfy a given number of users for our tested applications. Three StarTrack servers can support a peak load of around 120 requests per second (without caching) or up to 780 (with caching). Without caching, this allows over 5 million queries uniformly distributed over a period of 12 hours, corresponding to an average of 5 queries per user given a population of 1 million users requesting personalized driving directions.

In the case of ride-sharing, it’s desirable that track trees are pre-built and kept in memory. In order to create and cache a single or multiple track trees with each user’s top 5 tracks, a ride-sharing application satisfying 1 million users would require approximately 10 GB of memory. A single server holding all this data could allow a peak load of 35 requests per second, or more servers could be used if higher peak loads need to be handled.

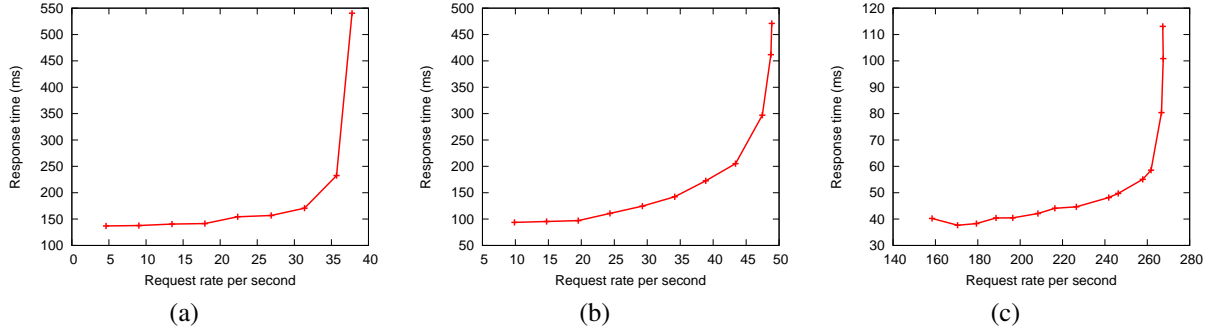


Figure 8: Response times for the RS and PDD applications under varying request rates. (a) RS application; (b) PDD where users’ tracks are not cached; (c) PDD where users’ tracks are previously cached.

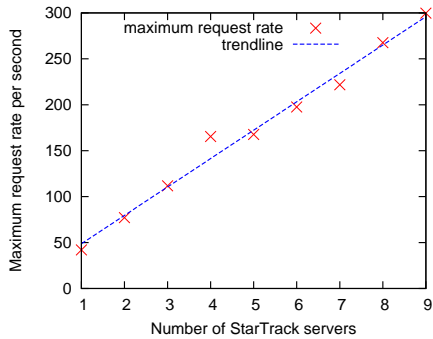


Figure 9: Maximum aggregate request rate with increasing numbers of StarTrack servers.

## 7 Related Work

As mobile devices have become equipped with the ability to determine their own location, there has been an emergence of applications that collect and utilize users’ location data. The research community has proposed a number of useful location-based applications. Traffic prediction [11, 24], ride-sharing [14], personalized driving directions [21] and electronic tour guides [1, 25] are some compelling examples.

At present, every application is forced to maintain its own silo of user location data. StarTrack addresses this problem by providing a common infrastructure that collects location information and enables access to it by multiple applications. In recent years, a number of data platforms (such as Twitter and Facebook) have emerged that enable sharing of information between users. These platforms provide external application developers with an API for accessing user information. StarTrack can be thought of as a platform that stores and enables access to the tracks traversed by users in their daily lives.

Efficient collection of location data is an important precursor to organizing this data and making it accessible. The CarTel project [13] is a distributed sensor net-

work that supports data collection from mobile phones and vehicular sensor networks. CarTel allows applications to visualize traces stored in a relational database using spatial queries.

Database researchers have extensively studied the problem of storing, indexing, and retrieving trajectories. A trajectory is similar to a track in our system and is modeled as a geometric object with 3 dimensions: two for geographical location and a third for time. Prior work has focused on range queries on trajectories and has led to novel indexing techniques. For example, research has shown that it is more efficient to separate the spatial and temporal dimensions and to first index the spatial dimensions [6]. There is also research that optimizes storage and query costs when trajectories are drawn from a fixed road network [2, 4, 22]. Some of the design decisions in StarTrack are based on similar observations. StarTrack additionally allows tracks with very similar geometries to be pruned, resulting in even greater savings. Furthermore, StarTrack exploits the repetitiveness in users’ tracks drawn from a road map to implement efficient similarity and common segment queries, which are not studied in previous work.

## 8 Conclusion

StarTrack enables a broad class of track-based applications, involving both individual users and social networking groups. Our original design of the StarTrack platform focused almost exclusively on the set of operations that would be useful to application developers and ignored performance and scalability considerations. Significant work went into revising the StarTrack design and implementation to enhance its efficiency, robustness, scalability, and ease of use. In some cases, we were able to apply well-known techniques, such as vertical data partitioning and chained declustering. However, most of the observed improvements come from innovative data structures like track trees, new representations for canonicalized tracks,

and novel uses of delayed execution and caching.

The end result is a track-based service that shows several orders of magnitude improvement in performance for operations that are commonly used in the applications that we have developed. This allows such applications to meet their scalability requirements. Moving forward, we plan to build and deploy additional track-based applications to further validate the practical utility of our redesigned service.

## Acknowledgments

We thank our former interns Ganesh Ananthanarayanan and Erich Stuntebeck for their work on earlier versions of the system. We thank Lenin Ravindranath and Renato Werneck for helpful discussions; John Krumm, Paul Newson, and Eric Horvitz for providing us with user location data and the map matching software; and Daniel Delling for the shortest path software used to generate synthetic tracks. The anonymous referees and our shepherd, Brad Karp, provided useful suggestions for improving the paper.

## References

- [1] ABOWD, G. D., ATKESON, C. G., HONG, J., LONG, S., KOOPER, R., AND PINKERTON, M. Cyberguide: A mobile context-aware tour guide. *Wirel. Netw.* 3, 5 (1997), 421–433.
- [2] ALMEIDA, V. T. D., AND GÜTING, R. H. Indexing the trajectories of moving objects in networks. *Geoinformatica* 9, 1 (2005), 33–60.
- [3] ANANTHANARAYANAN, G., HARIDASAN, M., MOHAMED, I., TERRY, D., AND THEKKATH, C. A. StarTrack: A framework for enabling track-based applications. In *MobiSys '09: Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (2009), pp. 207–220.
- [4] BRAKATSOULAS, S., PFOSE, D., AND TRYFONA, N. Practical data management techniques for vehicle tracking data. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (2005), pp. 324–325.
- [5] CAO, L., AND KRUMM, J. From GPS traces to a routable road map. In *GIS '09: Proceedings of 17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems* (2009), pp. 3–12.
- [6] CHAKKA, V. P., EVERSPOUGH, A., AND PATEL, J. M. Indexing large trajectory data sets with SETI. In *CIDR '03: 1st Conference on Innovative Data Systems Research* (2003).
- [7] CHEN, D., GUIBAS, L. J., HERSHBERGER, J., AND SUN, J. Road network reconstruction for organizing paths. In *SODA '10: Proceedings of 21st ACM-SIAM Symposium on Discrete Algorithms* (2010), pp. 1309–1320.
- [8] CHENG, Y.-C., CHAWATHE, Y., LAMARCA, A., AND KRUMM, J. Accuracy characterization for metropolitan-scale wi-fi localization. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (2005), pp. 233–245.
- [9] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of relational views. In *VLDB '78: Proceedings of the 4th International Conference on Very Large Data Bases - Volume 4* (1978), pp. 368–377.
- [10] FINKEL, R. A., AND BENTLEY, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (2004), 1–9.
- [11] HORVITZ, E., APACIBLE, J., SARIN, R., AND LIAO, L. Prediction, expectation, and surprise: Methods, designs, and study of a deployed traffic forecasting service. In *UAI '05: Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence* (2005), pp. 433–437.
- [12] HSIAO, H.-I., AND DEWITT, D. J. Chained declustering: A new availability strategy for multiprocessor database machines. In *ICDE '90: Proceedings of the 6th International Conference on Data Engineering* (1990), pp. 456–465.
- [13] HULL, B., BYCHKOVSKY, V., ZHANG, Y., CHEN, K., GORACZKO, M., MIU, A. K., SHIH, E., BALAKRISHNAN, H., AND MADDEN, S. CarTel: A distributed mobile sensor computing system. In *SenSys '06: Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems* (2006), pp. 125–138.
- [14] KAMAR, E., AND HORVITZ, E. Collaboration and shared plans in the open world: Studies of ridesharing. In *IJCAI'09: Proceedings of the 21st International Joint Conference on Artificial Intelligence* (2009), p. 187.
- [15] King county metro transit: Rideshare online <http://www.rideshareonline.com>, 2010.
- [16] KRUMM, J., AND HORVITZ, E. The Microsoft multiperson location survey. Tech. Rep. MSR-TR-2005-103, Microsoft Research, Redmond, WA, USA, 2005.
- [17] KRUMM, J., LETCHNER, J., AND HORVITZ, E. Map matching with travel time constraints. In *SAE '07: Proceedings of the Society of Automotive Engineers World Congress* (2007).
- [18] LANDIN, P. J. The mechanical evaluation of expressions. *Computer Journal* 6 (January 1964), 308–320.
- [19] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ASPLOS '96: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 84–92.
- [20] NEWSON, P., AND KRUMM, J. Hidden Markov map matching through noise and sparseness. In *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2009), pp. 336–343.
- [21] PATEL, K., CHEN, M. Y., SMITH, I., AND LANDAY, J. A. Personalizing routes. In *UIST '06: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (2006), pp. 187–190.
- [22] PFOSE, D., AND JENSEN, C. S. Indexing of network constrained moving objects. In *GIS '03: Proceedings of the 11th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems* (2003), pp. 25–32.
- [23] SONG, C., QU, Z., BLUMM, N., AND BARABSI, A.-L. Limits of predictability in human mobility. *Science* 327, 5968 (2010), 1018–1021.
- [24] THIAGARAJAN, A., RAVINDRANATH, L., LACURTS, K., MADDEN, S., BALAKRISHNAN, H., TOLEDO, S., AND ERIKSSON, J. VTrack: Accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (2009), pp. 85–98.
- [25] ZHENG, Y., WANG, L., ZHANG, R., XIE, X., AND MA, W.-Y. GeoLife: Managing and understanding your past life over maps. In *MDM '08: Proceedings of the 9th International Conference on Mobile Data Management* (2008), pp. 357–358.