StarTrack: A Framework for Enabling Track-Based Applications

Ganesh Ananthanarayanan[*]
Microsoft Research
Mountain View, California
ganesha@cs.berkeley.edu

Maya Haridasan
Microsoft Research
Mountain View, California
mayah@microsoft.com

Iqbal Mohomed
Microsoft Research
Mountain View, California
iqbal@microsoft.com

Doug Terry
Microsoft Research
Mountain View, California
doug.terry@microsoft.com

Chandramohan A. Thekkath
Microsoft Research
Mountain View, California
thekkath@microsoft.com

ABSTRACT

Mobile devices are increasingly equipped with hardware and software services allowing them to determine their locations, but support for building location-aware applications remains rudimentary. This paper proposes tracks of location coordinates as a high-level abstraction for a new class of mobile applications including ride sharing, location-based collaboration, and health monitoring. Each track is a sequence of entries recording a person's time, location, and application-specific data. *StarTrack* provides applications with a comprehensive set of operations for recording, comparing, clustering and querying tracks. *StarTrack* can efficiently operate on thousands of tracks.

Categories and Subject Descriptors

C.2 [**Computer-Communications Networks**]: [Distributed Systems]

General Terms

Design, Algorithms, Measurement, Performance

Keywords

Tracks, location aware, GPS location

1. INTRODUCTION

Providing rich support for *tracks*, recorded time-ordered sequences of visited locations, enables a new class of applications on mobile devices. Mobile devices that can determine their own physical location, utilizing GPS for instance, are already becoming commonplace, as are development platforms for building location-aware applications like navigation services, targeted advertisements, and social rendezvous. Current mobile platforms provide application developers with methods for retrieving the device's current location. The location can be obtained by a number of means, including GPS hardware, cell tower triangulation, and WiFi beacons. This paper explores expanding the set of location-oriented primitives to include the notion of a track as a new, fundamental abstraction.

Tracks play a prominent role in a variety of emerging mobile applications from trip planning to online games to fitness monitoring. Consider a ride sharing application, for example. By recording and comparing the tracks of commuters driving to and from work, this application suggests co-workers that can potentially drive together, thereby reducing pollution, gas consumption, and traffic congestion. Similarly, tracks left by others might help someone new to an area discover interesting biking trails, determine safe places to walk, or indirectly share experiences with friends and family.

StarTrack is a system that enables extensive operations on tracks. A track is a discrete and sampled representation of a continuous route. Mobile devices collect tracks and opportunistically upload them to a central server. *StarTrack* includes facilities for storing, comparing, clustering, indexing and retrieving tracks. It serves as the foundation for building large-scale track-based services.

Overall this paper makes the following contributions: First, we present an abstraction of a track that we believe is useful for a large class of interesting applications. Second, we present efficient algorithms for manipulating tracks including comparison and clustering. Third, we demonstrate that *StarTrack* shows encouraging results up to 10,000 tracks across multiple users. Also, since our operations are straightforward to partition, we can easily scale to larger numbers of tracks.

The rest of the paper is organized as follows. Section 2 presents an overview of the *StarTrack* framework including the system architecture, programming interface and motivating applications. The fundamental operations supported

[*]Work done during a summer internship at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSys'09, June 22–25, 2009, Kraków, Poland.
Copyright 2009 ACM 978-1-60558-566-6/09/06 ...$5.00.

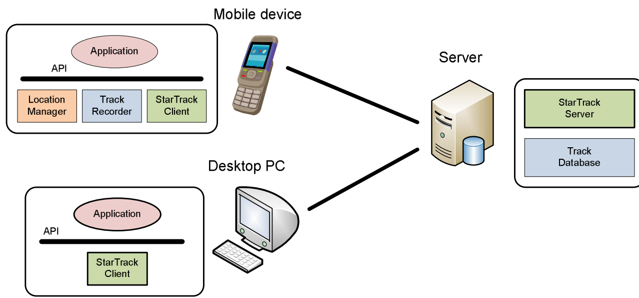


Figure 1: The *StarTrack* architecture

on tracks − comparison, clustering and retrieval − are described in Section 3. We present the results of our evaluation in Section 4. We contrast *StarTrack* with prior work in Section 5 and conclude in Section 6.

2. SYSTEM OVERVIEW

2.1 Key Concepts

StarTrack is a system that provides the building blocks for developing track-based applications. A track is a time-ordered sequence of *location* readings, each of which is a *track entry*. A track is intended to capture the path taken by a mobile device or, more importantly, a person in possession of a mobile tracking device. A captured track may differ from the actual path taken in two ways. First, the track contains only a subset of the points along the path since the device's location is sampled at some frequency rather than continuously. Second, the reported location may be an approximation of the actual location due to characteristics of the physical device and location estimation techniques.

Tracks are recorded, under control of an application, and saved in a database along with previously recorded tracks from various users. Each track is owned by a particular person, and the owner of a track can specify who is allowed to access that track. Each track entry is a tuple consisting of a location, time, and optional application-specific metadata in the form of an XML document with arbitrary contents. For instance, users may choose to attach photos, sticky notes, or location based advertising to particular track entries.

2.2 Architecture

The *StarTrack* system is based on a client-server architecture. Figure 1 depicts a system with a cell phone, desktop PC, and the *StarTrack* server. In practice, a system will contain many mobile devices and any number of stationary PCs interacting with a (logically) centralized server. This figure also shows the software components that run on both mobile and fixed devices as well as on the server.

Each participating mobile device is assumed to have a means of determining its current location through a local *location manager*. The location manager determines the device's location using an available localization technology, such as the Global Positioning System (GPS), GSM localization, Wi-Fi hotspots, etc. [26, 12, 31] This allows *StarTrack* to work with any localization scheme.

A *track recorder* running on each mobile device periodically retrieves its current coordinates from the location manager and saves this location along with the current time as a new track entry. An application running on the device determines when new tracks are started and when a track is completed. This application can choose to either discard or permanently save a recorded track. Tracks are stored in the device's local storage system until they can be uploaded to the server. *StarTrack* allows applications considerable leeway on when and how the tracks are uploaded to the *StarTrack* server. For example, applications on devices that are disconnected from the server, or that wish to wait until a better connection is available, can locally record their tracks and upload them in batches at a later time. In this paper we do not discuss the features of the location manager or the track recorder any further.

The application programming interface (API) provides applications with a comprehensive set of operations on tracks. Operations are implemented through a combination of client code running on a device and code running on the server where the tracks are stored. The *StarTrack* client forwards local calls to the server so that operations can be performed close to the data avoiding the overhead of communicating track-related data to the clients. However, the client could also cache data locally and perform some local operations, though we do not support this in our current implementation.

The *StarTrack* API is also available to applications that run on stationary devices, such as a desktop PC. For example, in our ride sharing application, tracks are captured on cell phones while people drive to and from work. However, the application that suggests commute partners need not run on the cell phone; this application more expediently runs on a user's PC where it can present the user with maps and other visual information.

The *StarTrack* server is implemented on top of SQL Server. Tracks are stored, indexed, and queried using SQL Server 2008's support for spatial data. Currently, the *StarTrack* server runs on a single machine. To provide a more scalable infrastructure, the track data could easily be shared, i.e. replicated and distributed, across multiple servers based on geographic regions, users, or applications. Our server code implements higher-level operations on tracks such as comparing and clustering tracks. The next section describes the operations on tracks that are provided through the *StarTrack* API, and Section 3 goes into more detail about the algorithms used.

The server enforces conventional access controls on tracks in order to protect the privacy of users. That is, associated with each track is a list of people who are allowed to access the track (and use it in other *StarTrack* operations). However, if all users treat their tracks as private, then applications such as ride sharing become much less functional. Moreover, the server itself must be trusted since it has access to all tracks. To alleviate this concern, *StarTrack* could adopt well-known techniques for cloaking track data [19, 16, 20].

2.3 Programming Interface

The operations in *StarTrack*'s API fall into five categories: recording, manipulating, comparing, clustering, and querying tracks. Table 1 lists a simplified set of key operations in this API. It is intended to provide the flavor of the API rather than a complete specification.

Although a device's track recorder could continually capture tracks and send them to the *StarTrack* server, this may not always be desired since computing a device's location can

Recording Tracks
trk = StartTrack();
EndTrack(trk);
SaveTrackEntryMetadata(trk, metadata);
SaveTrack(trk, metadata);
RemoveTrack(trk);
Manipulating track
trk = ClipTrack(trk, region);
trk = SpliceTracks(trk1, trk2);
entries = ExtractTrack(trk);
entry = GenerateTrackEntry(coords, time, metadata);
trk = GenerateTrack(entries, metadata);
Comparing tracks
similarity = CompareTracks(trk1, trk2, constraints);
boolean = SimilarTracks(trk1, trk2, constraints);
Clustering tracks
trks = ClusterTracks(trks);
trk = RepresentativeTrack(trks);
Querying tracks
trks = QueryTracksByRegion(region);
trks = QueryTracksByOwner(owner);
trks = QueryTracksByMetadata(metadata);
trks = QueryTracksByTime(start, end);
trks = QueryTracksByTrack(query-trk);

Table 1: Operations on tracks

be computationally expensive and consume precious energy. Moreover, each application may have distinct notions of what tracks they want captured. Thus, *StarTrack* lets applications running on a device indicate when tracking is desired by calling the *StartTrack* and *EndTrack* operations. For example, a device's GPS hardware need not be turned on until *StartTrack* is called, saving battery life. Upon calling *SaveTrack*, the currently recorded track is stored in the device's persistent storage, and later is uploaded to the server. The application can also call *SaveTrackEntryMetadata* at any time while a track is being recorded to associate application-specific metadata with the device's current location.

A number of operations manipulate tracks in various ways. For instance, *ClipTrack* reduces a given track to the segment that lies in a given geographical region. *SpliceTrack* connects the endpoint of one track to the beginning of another, producing a new track. Operations exist for extracting entries from a given track. Synthetic tracks can be generated and added to the database if so desired.

One of the main operations in *StarTrack* is *CompareTracks*, which returns a measure of the similarity between two tracks. The returned similarity metric is a real number between 0 and 1. Larger numbers, i.e. metrics closer to 1, indicate a higher degree of similarity between the two tracks. Conceptually, the comparison returns the fraction of entries in common between the two tracks. Optionally, constraints can be specified to indicate that one track should be a subset of the other, the tracks should have the same starting or ending point, and so on. A variant of the compare operation, called *SimilarTracks*, returns a Boolean value indicating whether the two tracks are sufficiently similar, that is, have a similarity above some meaningful threshold.

The *ClusterTracks* operation takes a set of tracks and groups them such that similar tracks are placed in the same cluster. It then selects a *representative track* from each cluster and returns all of these representative tracks. In practice, people tend to visit the same places regularly and follow the same path, potentially generating a large number of similar tracks in the database. Clustering can be used to eliminate duplicates. The *RepresentativeTrack* operation is similar to *ClusterTracks* except that it returns a single track, the representative track from the largest cluster.

Finally, the API includes a number of operations for running queries against the track database. Each of the queries shown in Table 1 operates on the complete track database and returns all of the tracks that match the query. They allow applications to ask for tracks that cross a certain geographical region, that belong to a particular user, that contain specific metadata, that were captured in a given time span, or that are similar to a given track (according to the *CompareTracks* operation). Variations on these basic query operations (not shown in the figure) return database cursors that can be used to retrieve selected tracks one at a time. Such cursors also can be passed to other query operations in order to compose complex queries involving multiple criteria. Queries can return results in a desired order. For instance, when querying for tracks that match a given other track, the application may want the results sorted in decreasing order of similarity.

2.4 Sample Applications

Location-based applications can be roughly classified into three categories that are defined by the type of data that is available to them: a person's current location, past locations, or tracks. The last two categories require access to a database of either past locations that have been visited by a person or previous routes taken by the person (as studied in this paper). An additional dimension of applications is characterized by whether the data made available to the application pertains to a single person, the person running the application, or multiple people, such as a social network. This yields six main categories of location-based applications, as depicted in Table 2 with examples of each category. The most common mobile applications, the first row of this table, are those that make use of a person's current location. These include applications that recommend nearby services, provide driving directions, guide tourists [10], or locate friends [13]. The second row summarizes applications that present or process information about past locations that have been visited by a person and his associates. For example, recorded locations can be collected into a person's digital diary [14]. This paper focuses on support for the third row of this table, personal and social track-based applications. To further motivate *StarTrack*, we present some specific applications involving tracks; these applications use a wide variety of *StarTrack* operations as indicated in the following discussion.

Ride Sharing: There is significant need to conserve energy across different aspects of our lives. Since transportation is a major energy consuming activity, a system that automatically suggests potential partners for ride sharing based on tracks can be very useful. Figure 2 presents the high-level code for a ride sharing application built using *StarTrack*. This code can run on any device that has a network connection to the *StarTrack* server, such as a user's PC. The device-resident code that records tracks is not shown since it is straight-forward. In particular, it calls *StartTrack*

	Personal	Social Network
Current location	nearby restaurant recommendations, driving directions, automated guides	friend finder, rendezvous, social awareness, urban games
Past locations	digital diary, personal travel journal, augmented memory	recommender systems, post-it notes
Tracks	advertisements, health monitoring	ride sharing, urban sensing, collaboration, discovery, shared experiences, common interests

Table 2: A taxonomy of location-based applications

```
// get a representative track for my own commute
myTracks = QueryTracksByOwner("me");
myMorningTracks = QueryTracksByTime(myTracks, "7:00 am", "9:00 am");
myCommuteTrack = RepresentativeTrack(myMorningTracks);

// find tracks of others with similar commutes
similarTracks = QueryTracksByTrack(myCommuteTrack);
for each track in similarTracks do
  person = GetTrackOwner(track);
  personTracks = QueryTracksByOwner(person);
  personMorningTracks = QueryTracksByTime(personTracks, "7:00 am", "9:00 am");
  personCommuteTrack = RepresentativeTrack(personMorningTracks);
  if SimilarTracks(myCommuteTrack, personCommuteTrack) then
     report person as a potential ride sharing partner
  endloop;
```

Figure 2: Ride sharing application on *StarTrack*

whenever a person starts driving in the morning and calls *EndTrack* followed by *SaveTrack* when the person arrives at work; similarly, it creates a new track for the person's drive home from work.

The procedure in Figure 2 first determines the commute track for me, the person who is looking for ride share partners. It does this by first retrieving all my tracks and narrowing this set to morning tracks. It then calls the *RepresentativeTrack* operation to cluster my tracks and return the representative track from the largest cluster. This representative track indicates my typical commute. To find others with similar commutes, the *QueryTracksByTrack* operation is called with my representative track as the parameter. Unfortunately, this returns tracks for anyone who has ever driven a similar path, even if someone does not usually take that route. To determine suitable commute partners, tracks for each of the potential partners are retrieved and clustered to determine their typical commute. Finally, for each potential partner, the person's representative track is compared to mine to ensure that we really do frequently drive the same route.

Urban sensing: Participatory and location-based sensing in urban settings has received attention particularly by the research community. The underlying theme in these projects involves collecting, aggregating and analyzing *sensor* data using mobile devices [32, 21] or specialized hardware [22] and tagging the data with the current location. Examples of such data in prior work include traffic congestion [22, 32], particle pollution readings [21], noise levels and visibility, and signal characteristics of Wi-Fi [22], cellular and Wi-Max. Since *StarTrack* facilitates easy analysis, querying and presentation of location-tagged data it can help build distributed sensing applications.

Advertisements: Location-based advertisements are being investigated and deployed with significant interest by corporations because of their immense economic potential. One of the popular usage scenarios involves a user being given a special coupon or discount when they are in the vicinity of a store (known as proximity marketing). Tracks provide a richer semantics for advertisers to create customized incentives for potential customers. For instance, a furniture store in a mall can offer a customer a 10%-off coupon, when the customer has visited two of its rivals on the other side of the mall.

Collaboration: Collaborative applications among mobile devices often require a way for the devices to rendezvous. For activities like collaborative downloading [11] or multiplayer games [34, 25] among devices in public transit vehicles, it would be beneficial to find peers that are likely to produce minimal disruption. By comparing past tracks, applications can predict likely collaborators, which in turn can be used for planning and scheduling activities.

Discovery: If someone is new to an area, prior tracks might help them learn something. For example, one of the questions someone often has when in a new city is whether it is safe to walk around the streets at night. A person could answer this question by querying for tracks in the area of his hotel in which people were walking after 10 pm. Another example is looking for good places to ride a bike or safe streets on which to ride a bike during commute hours. Again, these could be determined by running queries for sets of "bike" tracks.

Shared experiences: Imagine a shopping mall in which teenagers wander around during the weekend. One use of location based services is to inform a teenager when their friends are also in the mall at the same time. However, even

if friends go at different times, one might want to see the tracks that their friends took, e.g., what stores they visited, so that when they see each other on Monday they can talk about the cute clothes they saw or the sale at a particular store.

Common interests: One major benefit of having a track database is being able to find tracks taken by people with similar interests or characteristics. A person may not want to know which tracks where taken by the population at large, but rather which were taken by friend, colleagues or fellow hiking enthusiasts. As an example, we have built and used a trade show application on top of the StarTrack infrastructure in a conference visited by over five thousand employees at Microsoft. The application allows visitors to check not only their tracks (paths of booths that they have previously visited), but also to identify popular tracks, find people with similar track histories, and get recommendations on other booths to visit based on the booths visited by other visitors with similar tracks.

Health monitoring: Devices that constantly monitor a person's bio-stats are becoming popular, especially among the elderly, as are devices that record activities such as walking or jogging [3]. Associating readings with time and location, i.e., with track entries, can allow health practitioners to more effectively mine a person's vital data.

3. FUNDAMENTAL OPERATIONS

Recall from Section 2 that a track is a time-ordered sequence of location readings. *StarTrack* represents locations as latitude/longitude coordinates, though we have also considered accommodating cell-tower and Wi-Fi access point fingerprints. *StarTrack* supports several operations on the tracks as outlined in Section 2. In this section we describe three key functions that *StarTrack* provides: track comparison, track clustering, and track retrieval.

3.1 Track Comparison

The track comparison function enables an application writer to determine if two paths represented by their track entries are "similar" to each other. In its simplest form, the track comparison operation compares the location fields in the track entries of two tracks and returns a value between 0 and 1, which quantifies the *similarity* between the paths that the tracks represent.

Spatial comparisons alone may not be sufficient for some applications. For instance, for a user to provide a ride to another, there must be a temporal match in addition to a spatial match (i.e., the users must be traveling to approximately the same destination at approximately the same time). *StarTrack* performs both spatial and temporal matching in sequence: It inputs the set of tracks matched with one criterion to the second criterion. The actual order is not significant. We support simple bounds on the time differences between track entries in temporal comparisons (e.g., find two tracks that are within 30 minutes apart of each other). In the remainder of the section, due to space restrictions, we focus on the details of the spatial comparison and do not discuss temporal comparisons.

Track comparison is a harder problem than is apparent at first blush because it is not just a matter of matching two sets of track entries to see if the paths are similar. Recall that a track is a sampling of a path and is therefore only an approximation of it. Multiple factors including the sampling frequency of track entries and errors in location determination can lead to dissimilar sets of track entries for two identical paths.

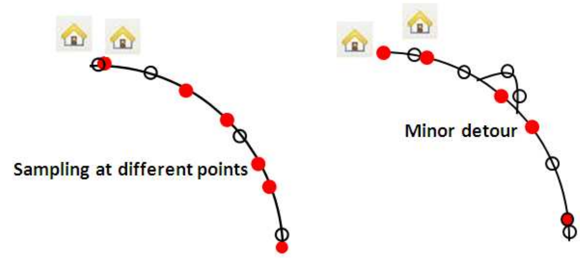


Figure 3: Track comparison is hard due to sampling differences and detours. The "home" icon represents the start of each track

(a) Sampling Variability: The points sampled in a track are a function of the sampling frequency and the speed at which the mobile device is traveling.

The sampling frequency can change over time because GPS receivers are known to be power-hungry [32] and an application may prefer a lower sampling rate when the mobile device's battery is low. Lower sampling frequencies result in a sparser set of track entries than otherwise.

Differences in the speed of motion (e.g., walking at 3 mph or cycling at 15 mph or driving at 40 mph) also cause variations in the track entries. Furthermore, terrain (e.g., a steep hill) or traffic congestion can also cause noticeable speed variation. Thus even with a fixed sampling frequency, we can end up with dissimilar track entries for a path. This is illustrated in Figure 3.

(b) Minor Detours: Minor detours from a path get reflected in the resultant tracks (see Figure 3). For example, detours can easily occur due to unexpected traffic congestion or because the user is running a small errand. We would like our track comparison algorithms to be not overly sensitive to these detours because some applications require that the comparison algorithms be sufficiently robust against this type of user behavior.

(c) Subsets: One complication in track comparison arises due to sub-matches, i.e., when the path represented by one track is included as a subset in the path represented by another track. Figure 4 illustrates this scenario. The track marked by open circles is a subset of the track denoted by solid points. Consider the applications of ride-sharing and urban sensing. Clearly, the person in the track with solid points can give a ride to the person in the other track. Likewise, urban sensing data collected by the person in the track with open circles can be used to alert the person in the other track. This consideration leads us to design our comparison algorithm to be *asymmetric*, i.e., for two tracks T_a and T_b, $Similarity(T_a, T_b) \neq Similarity(T_b, T_a)$.

(d) Intersecting Segments: As shown in Figure 4, tracks that have no similarity in their start and end points could still have intersecting segments. A comparison algorithm biased in favor of start and end track entries is likely to output a low similarity score for such tracks. Many applications, particularly ones that depend on ad-hoc collaboration, are likely to care about the presence of a sufficiently long common segment without necessarily caring about the

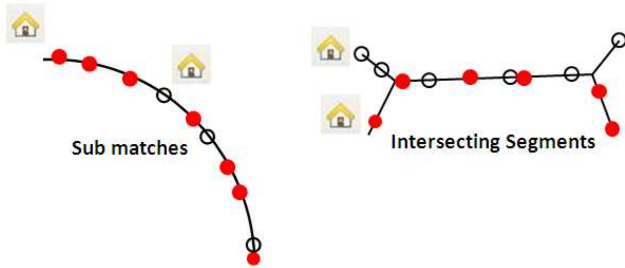


Figure 4: Track comparison is hard due to sub-matches and intersecting segments. The "home" icon represents the start of each track. Notice that in the second case, the tracks have a very long common segment that can be used by collaborating applications

start or the end of the paths, and may not benefit from an algorithm that favors the start/end points.

The observations cited above lead us to track comparison algorithms that respect the following criteria:

- approximate or fuzzy matching of track entries
- comparison over all track entries with equal weight
- asymmetry

Note that applications can modify these properties according to their requirements using the *constraints* argument in *CompareTracks*.

Our algorithms are based on the location field in the track entries. Our comparison algorithms define an area around one track and find the fraction of points in the other track that fall inside this area. This fraction is the similarity value. We have explored comparison schemes where the shape of the area chosen is a circle or a strip whose dimensions vary depending on the relative positions of the points that make up the track. Readers familiar with statistical filtering techniques (e.g., Kalman filters [24]) will notice that our scheme is a degenerate case of applying statistical filtering to two tracks, which can be treated as sampled observations of a trajectory in 2-dimensions. Also, a simple probabilistic model on which to base the statistical approach is readily available depending on the shape of the area selected.

We recognize that similarity is often an application specific concept and intend to make provisions to accommodate user-defined similarity functions. For now we discuss two predefined similarity functions implemented in the system. In the rest of this section we denote the candidate tracks to be compared as T_a and T_b.

3.1.1 Circle-based Algorithm

We define a circle of a certain radius, r, around each of the points in T_a and find the fraction of points in T_b that fall inside any of the circles (see Figure 5).

Since the track entries are time-ordered we use this information to ensure the *direction* of the tracks is the same. If a track entry in T_b falls within the circle for some entry in T_a, then other entries in T_b with later timestamps should fall within the same circle or circles corresponding to later entries in T_a (or else lie outside of any circles). We call this

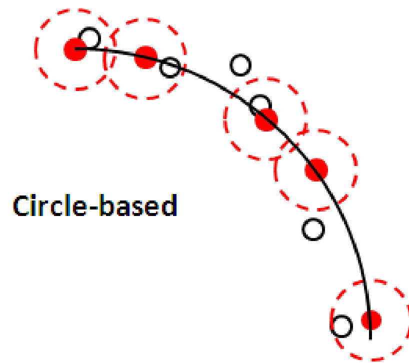


Figure 5: Circle-based track comparison algorithm

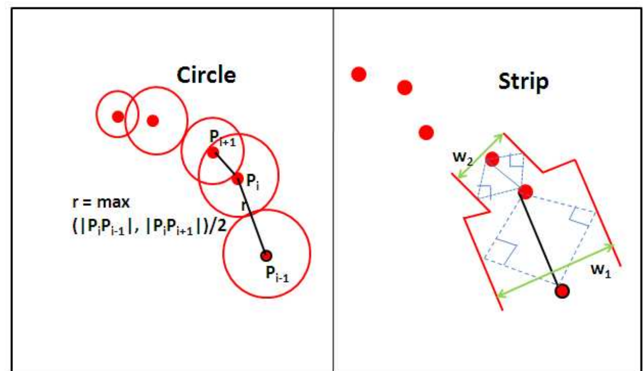


Figure 6: Variable estimation of circle radius and strip width for track comparison

the time monotonicity property. We do not place strict requirements on the monotonicity to account for measurement errors associated with GPS readings; a small percentage of points in T_b are allowed to violate the monotonicity requirement.

Estimating the radius: The radius of the circle, r, determines the accuracy of the estimation of similarity. A larger radius may result in higher similarity values while a smaller radius may lead to lower similarities. Thus, a poorly chosen radius can result in false-positives or false-negatives from an application's perspective.

The objective of the circles is to account for the variations at the points at which the track entries are sampled. Hence, they should ideally cover the area between the track entries, thereby re-constructing the underlying path. As an alternative to using circles with fixed radius, we use also consider varying values for r depending on the distance between consecutive points. For a given point P_i whose adjacent points are P_{i-1} and P_{i+1} (in the time-ordered sequence), the value of r is half of the maximum of the distances between its adjacent points as shown in Figure 6.

3.1.2 Strip

This scheme constructs a strip or band around track T_a of width w and calculates the fraction of track entries in track T_b that fall inside this strip (see Figure 7). For each pair of consecutive points we construct the strip by using rectangles

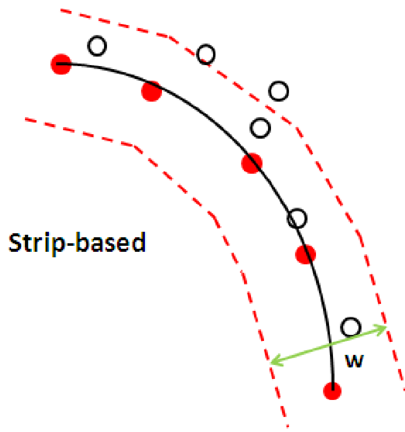


Figure 7: Strip-based track comparison algorithm

of width w, and length equal to the distance between the consecutive track entries.

For the same value of $2r$ and w, the strip-based comparison scheme is expected to be more accurate than the circle-based scheme, which can miss points that are close but just outside the circles. On the other hand, the strip-based algorithm involves more computation and is slightly more inefficient. As with the circle-based scheme, the strip-based comparison roughly enforces the time monotonicity property.

Estimating strip width: The width of the strip w impacts the similarity value returned by the comparison algorithm. Therefore, choosing a fixed width may not be ideal.

The width may be varied to capture the maximum possible curvature of the path between consecutive points. If we assume that such maximum possible curvature of the path subtends at least a right-angle, we end up with a rectangle between consecutive points where the width of the strip is set to the distance between the points. This is illustrated in Figure 6.

Given that there are tradeoffs associated with the use of different comparison algorithms, the *StarTrack* API provides flexibility when choosing between the circle or strip approaches, and between using fixed or variable parameters. We allow applications to specify minimum and maximum circle radii/strip widths in accordance with each application's semantics. A fixed circle radius or strip width may be imposed by setting these minimum and maximum values to be the same desired value.

3.2 Track Clustering

Users typically visit a small set of places via predictable routes as part of their daily routine (e.g., they travel between home and work, they go grocery shopping, etc.). This results in the *StarTrack* server storing multiple tracks of what is essentially the same path. Through track clustering, *StarTrack* allows applications to eliminate near duplicate tracks and group tracks into a smaller set of representative tracks. Track comparisons, as described in the previous section, can then be done against these representative tracks. This saves on computation costs as well as track retrieval times. Applications such as the ride sharing application, may choose to cluster the tracks belonging to individual users.

There are well-known techniques for clustering or grouping data items, two of which are *k-means* [18, 33] and *k-medians* [15]. Both of these techniques partition a group of n points into k groups, such that the distance (or the dissimilarity) between points in a group and a designated "cluster center" in the group is minimized. To use the k-means algorithm, it is necessary to assume that the points belong to $\mathbb{R}^d$, an assumption that is problematic in our case because our "points" are tracks, rather than points in euclidean space. Instead we use the k-medians approach, which is more tractable. In both algorithms, k is an input parameter: i.e., some external agent decides *a-priori* the number of clusters the algorithm is going to yield.

Our choice of k-medians has two potential pitfalls. The first issue is that k, the number of clusters need to be chosen before the algorithm can be run, and this is inherently difficult. *StarTrack* side-steps this issue by allowing application writers to input a suitable value of k, but it is still a potential shortcoming. There are algorithms related to k-medians that can determine the approximate value of k and cluster the n points into k clusters. We are considering implementing one such algorithm called *x-means* [17].

A second issue is that these clustering algorithms are not incremental, i.e., the entire input set must be available to the algorithm. This issue affects scaling. The ride sharing application, for example, deals with this by batching the k-median calculation into epochs. Each new track added for a user during any given epoch is simply assigned to one of the existing clusters that is most similar to it. Periodically, when a sufficient number of new tracks have been added, or when a certain amount of time has elapsed since the last epoch, the application triggers a new epoch by recalculating k-medians for all the tracks for that user.

3.3 Track Retrieval

Our current implementation of *StarTrack* utilizes SQL Server 2008 as its backend datastore. SQL Server supports spatial datatypes [5] and provides efficient querying via mechanisms such as multi-granularity spatial indices [4]. Spatial indices partition a given region into a set of grids, and the database keeps track of the association of rows in a table to each of the grids. Having multiple levels of granularity for such a spatial indexing system can help narrow down the set of candidate grids, and allows for faster retrievals.

Tracks are stored in SQL Server in a straightforward manner. A track corresponds to a single row in a table, and has a track identifier, XML metadata, track entry locations, and track entry timestamps. We store the locations and timestamps separately as it simplifies how we write queries.

The *StarTrack* API supports multiple ways to query tracks (such as by region, time or metadata). With one exception, the API calls are directly translated into SQL statements, and run against the database. The exception is an API call that provides querying by track. The semantics of this function are to return all tracks that are similar to the target track. We implement this by first finding a set of regions that surround the target track, retrieving all tracks that go through any of these regions, and then ordering the retrieved tracks based on the *CompareTracks* API function. The API also allows constraints to be placed on the result set, such as requiring proximity to the start and/or end point of the target track, or requiring a minimum overlap with the target track.

Track Name	Description	Distance (miles)
T-Stanford*	Drive around Stanford University	4.1
T-SF*	Drive from San Francisco to Mountain View	37.2
T-CT-1	Train from San Francisco to Mountain View	38.2
T-CT-2	Train from San Francisco to Redwood City	19.1
T-101	Drive from San Carlos to Mountain View on 101	18.1
T-280	Drive from San Carlos to Mountain View on 280	25.2
T-Rnd	Random track	2.5

(* − Two tracks of this type)

Table 3: Details of real tracks collected by volunteers

During our evaluation, we observed good performance for the various track retrieval functions (detailed in the evaluation section). While we utilized SQL Server's indexing system, we did not attempt to optimize the layout of data in the database. As such, further performance gains may be possible. Additional scale can be obtained by well-known database techniques such as partitioning and replication. *StarTrack*'s datastore can be partitioned in a straightforward manner by using geography.

4. EVALUATION

In this section, we evaluate the set of operations provided by *StarTrack*, focusing on track comparison, track clustering and track retrieval operations. All our experiments were performed on a 32-bit Windows Vista PC, with an Intel Core2 Duo CPU 3.00 GHz with 4.00GB of RAM. Our experiments are based on two sets of tracks, as described below.

Real Tracks: We collected nine real tracks from volunteers driving or commuting with GPS devices in the San Francisco Bay Area, each consisting of location entries collected within 30 second intervals. The collected set of tracks contains representative similar and disjoint paths, as well as sub-matches and super-matches. Table 3 describes the tracks and their notations.

Synthetic Tracks: With the goal of performing a large-scale evaluation of *StarTrack*, we used a realistic model to synthetically generate thousands of tracks in the Bay Area. Our track generation algorithm uses the set of road intersections in the Bay Area, and simulates differences in sampling frequencies and measurement errors.

We rely on an implementation of the shortest path algorithm proposed by Goldberg et al. [9], which computes the shortest path between any two points in the Bay Area and returns the set of traffic intersections falling in the path. Given the set of intersections, we assume that the path between two points is composed by the straight lines connecting consecutive intersections. Our track generation algorithm randomly picks points on or around one such hypothetical path, and uses them as track entries for a synthetic track.

We vary two parameters when choosing random points along a path, with the purpose of simulating variable sampling frequencies and measurement errors. These parameters allow us to compare and explore the tradeoffs among the different comparison algorithms. *Hop distance* is the distance between two consecutive entries in a track and accounts for variability in the axis parallel to the path. In our experiments, we vary the hop distance between predefined lower and upper bounds. The second parameter is the *measurement error*, which is the variation of the location reading in the axis perpendicular to the road. The *measurement error* for each track entry was randomly chosen between 0 and 100 meters. Figure 8 illustrates these parameters on a sample track.

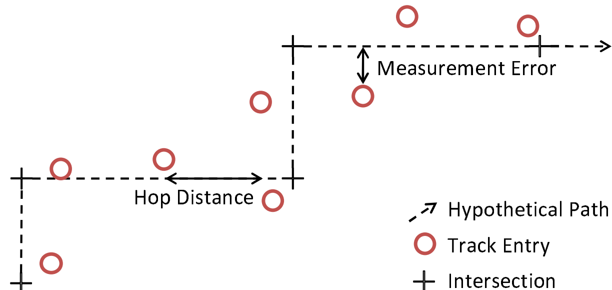


Figure 8: Synthetic track generation

Tracks	Ground Truth	Similarity	
		Circle	Strip
T-Stanford	Match	0.95	0.93
T-SF	Match	0.9	0.96
T-CT-1, T-CT-2	Sub-match	0.82	0.92
T-101, T-SF	Sub-match	0.83	0.93
T-SF, T-101	Super-match	0.44	0.38
T-Rnd, T-Stanford	No Match	0.09	0.03
T-280, T-SF	No Match	0.11	0.02

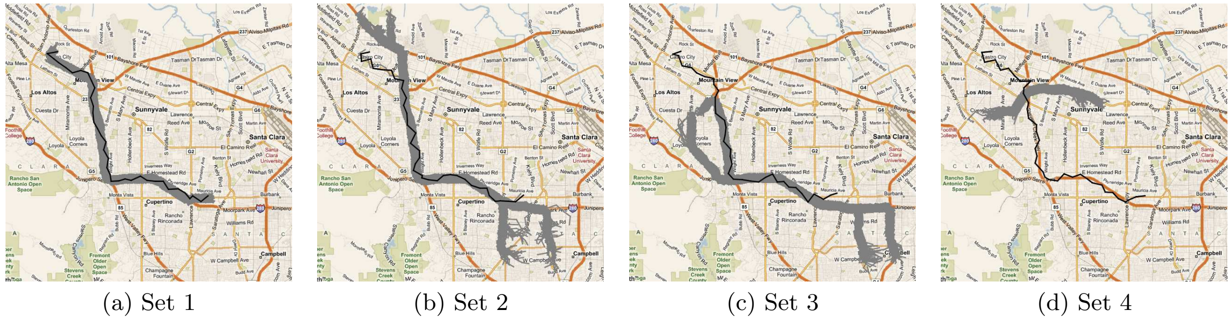
Table 4: Track comparison results on real tracks

4.1 Track Comparison

For an initial validation of the circle and strip-based approaches for track comparison, we evaluated their behavior when computing similarity between pairs of our set of real tracks. Table 4 shows the results of our experiments. Our ground truth was based on visual inspection of the tracks in a map. We observe high values of similarity values between tracks that visually overlap and very low similarity values between dissimilar tracks. We also note the asymmetry of our comparison schemes when comparing T-101 and T-SF: the similarity metric reflects that T-101 is a sub-match of T-SF (while T-SF is a super-match of T-101).

4.1.1 Parameter Selection: Fixed vs. Variable

We performed more systematic detailed experiments on a set of synthetic tracks to evaluate the performance of the circle and strip comparison approaches when we use fixed versus variable diameters/widths. The choice between these two approaches is left to application writers.



(a) Set 1 (b) Set 2 (c) Set 3 (d) Set 4

Figure 9: Track t (the thin black line) and four sets of tracks used in experiments

We first picked a single track t starting from an address in Cupertino, CA, to an address in Mountain View, CA, and compared it to four sets of tracks in the Bay Area (Figure 9). The choice of t was aleatory and we do not expect our general results to vary for different choices of t.

Each of the four sets contains 1000 tracks between particular start and end point regions. Differences between tracks within each group are due to slight differences in the start and end points, and mainly due to our artificially inserted sampling and measurement errors. Our goal with this set of experiments was to evaluate the effect of such errors on the computed similarities.

One of the advantages of using an artificially generated set of tracks is that it allows us to automatically compute the ground truth information about track similarity. We computed the *expected similarity* between two tracks based on the distances between intersections in the original paths used to create the tracks, which allows us to compute the length of common sub-paths as well as the total length of the paths. Table 5 presents the expected similarity between t and each of the four sets of tracks.

Set ID	Path	Expected Similarity
Set-1	Identical to t	1.0
Set-2	Cupertino to Mountain View	0.687
Set-3	Campbell to Los Altos	0.312
Set-4	Sunnyvale to Los Altos	0

Table 5: Details of synthetic tracks

We first fixed the diameter of circles and the width of strips used in the similarity computations to 200 meters each. In this experiment we set the average hop distance of all tracks to 100 meters. Figure 10 shows that the computed similarities were close to the expected values when using both the fixed circle and the fixed strip approaches (The error bars show the minimum and maximum variation). The *strip* approach performs better than the circle approach, given it has a better ability to extrapolate paths within track entries.

The previous experiment did not consider the effect of variable hop distances in tracks, which make it difficult to choose a fixed diameter/width. To deal with tracks with variable hop distances, *StarTrack* uses variable sized circles and strips, where the diameter/width is proportional to the distance between consecutive track entries in a track, as previously shown in Figure 6.

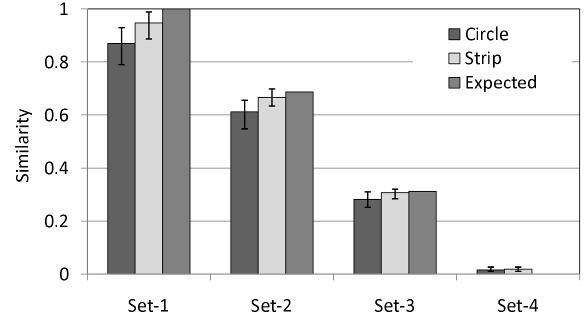


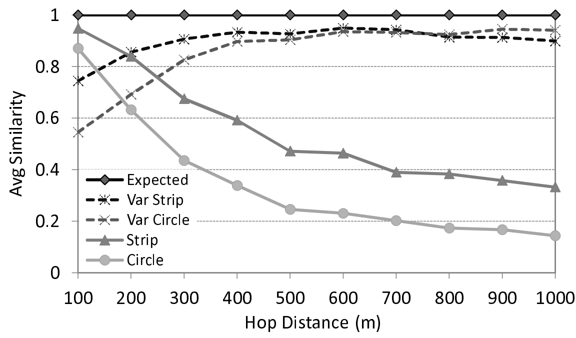
Figure 10: Track comparison results on synthetic tracks

In our next experiment, we compared the similarities computed using circles with a fixed diameter, and strips with a fixed width, circles with variable diameters and strips with variable widths. Figures 11(a) and 11(b) show the expected and average computed similarities for Set-1 (identical tracks) and Set-2 (partial match) when the average hop distance of the tracks is varied between 100 and 1000 meters, in increments of 100 meters. Observe that as data becomes sparser, fixed circles and strips (shown by the lines annotated with *Circle* and *Strip*) tend to progressively underestimate the similarity of tracks. Using variable parameters better accounts for data sparsity, providing similarity values closer to expected (lines annotated with *Var Circle* and *Var Strip*).

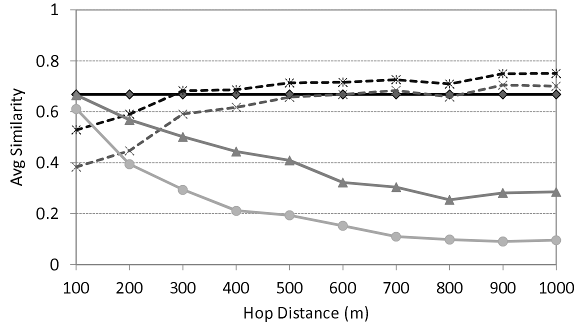
4.1.2 Evaluation with a Large, Diverse Set of Tracks

We also considered a larger set of random tracks containing ten thousand synthetic tracks around the South Bay Area. Our goal with this experiment was to evaluate the efficacy of the variable circle and strip approaches under more diverse settings. Again, we compared one fixed track t against every other track and computed the expected similarities. Most tracks in the set are disjoint from t, but the set was chosen so that there was a good number of partially matching tracks including some almost identical tracks. This can be observed in Figure 12, where we present the distribution of the expected similarity values.

Since the expected similarities now range anywhere between 0.0 to 1.0, instead of directly presenting the computed similarity values, we present the error of the computed values. The error is defined as the absolute difference between the expected similarity and the computed similarity of each generated track. We present the average errors, with error



(a) Set 1 - Identical



(b) Set 2 - Partial Match

Figure 11: Track comparison results when varying average hop distance of tracks

bars showing the minimum and maximum variations of the error.

One important detail to observe when using variable diameters/widths is that small distances between entries in a track can lead to excessively small circles and strips, unfairly lowering the similarity of tracks. This can be observed in our previous experiment, where Variable Circle and Variable Strip do not perform well for tracks with small hop distance values. To avoid this problem, an application may define a minimum value for the diameter of circles or width of strips based on the application's semantics.

Figures 13(a) and 13(b) show the similarity errors when comparing t to all other tracks. The two curves in each graph show the results when we use variable diameters/widths without enforcing a minimum value (*Without Minimum Diameter/Width*), and when we use variable diameters/widths but enforce a minimum value of 200 meters (*With Minimum Diameter/Width*). As expected, the enforcement of minimum diameters and widths improves results when the average hop distance is smaller.

4.1.3 Computational Time

The computational costs of comparing tracks should also be considered when deciding between the proposed approaches. Figure 14 presents the average times for computing the similarity of pairs of tracks when we increase the number of location entries per track. Considering that in the worst case both the circle and strip approaches roughly require comparisons between every pair of entries of the two compared tracks, the computational time is expected to grow quadratically as the number of entries in the compared tracks increases. The graph also shows that using variable diameters/widths does not incur any significant computational costs, compared to the fixed approaches.

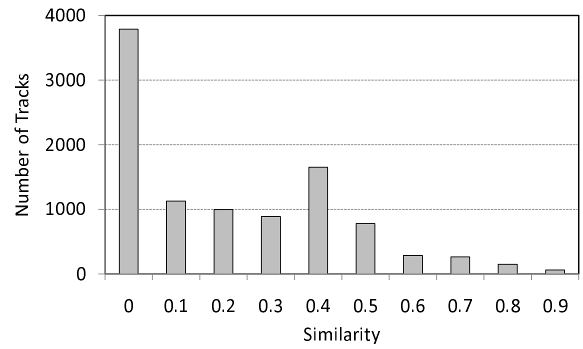


Figure 12: Distribution of expected similarity values between a fixed track t and other ten thousand generated tracks

4.2 Track Clustering

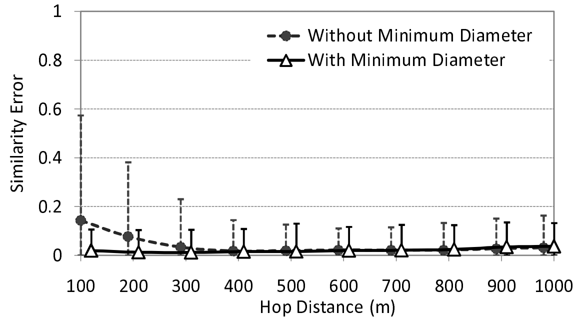
We now evaluate the performance and suitability of using a k-medians algorithm for grouping tracks into clusters and returning a set of k representative tracks, one from each cluster. As described in Section 3.2, our main goal with clustering is to eliminate near-duplicates.

Clustering requires a metric for qualifying the choice of a set of tracks, *rep tracks*, as the representative tracks for the larger group of tracks *all tracks*. In this case, we define the aggregate similarity as the sum of the similarities between each track in *all tracks* and its closest track in *rep tracks*. For clustering, the best set of representative tracks is the one that yields the highest aggregate similarity.

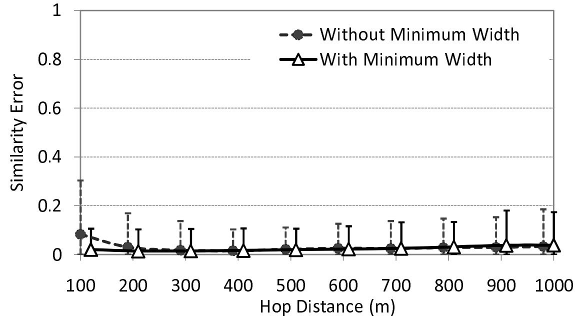
A naïve implementation of the k-medians algorithm considers all possible combinations of k representative tracks and computes the aggregate similarity for each in order to choose the best combination. Given that the number of combinations to be considered grows exponentially with the number of tracks, this clustering algorithm quickly becomes computationally unfeasible. Considering that an average comparison between two tracks requires 20 milliseconds, clustering 100 tracks would take over 17 days to execute.

Instead of considering all possible combinations, we used a version of the k-medians algorithm which uses an iterative refinement heuristic and provides an approximation of the best set of representative tracks and clusters. In each iteration it starts with randomly chosen combinations of k representative tracks until it either finds one that yields an aggregate similarity higher than a particular threshold or a predetermined number of combinations has been considered. Once a combination is chosen, the algorithm then proceeds to replace individual representative tracks locally, that is, tracks most similar to the ones in the current combination are chosen as substitutes until the aggregate similarity cannot be further improved.

We further optimized the performance of our clustering algorithm by observing that the choice of combinations (of representative tracks) to be explored could be pruned. By picking random combinations in the first phase of each iteration, there is a risk of considering combinations containing very similar tracks, which are obvious bad choices. The cost



(a) Variable Circle



(b) Variable Strip

Figure 13: Error between computed and expected similarities (between a track t and ten thousand random tracks when using variable sized circles/strips)

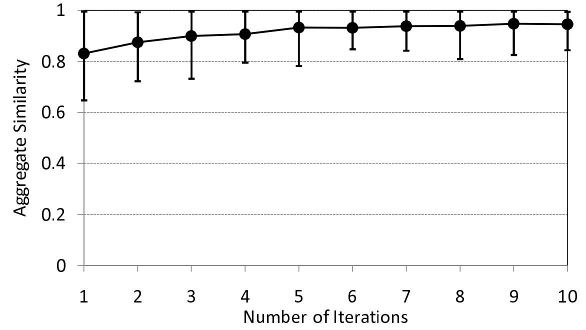


Figure 15: Performance of k-medians heuristic for clustering 100 tracks

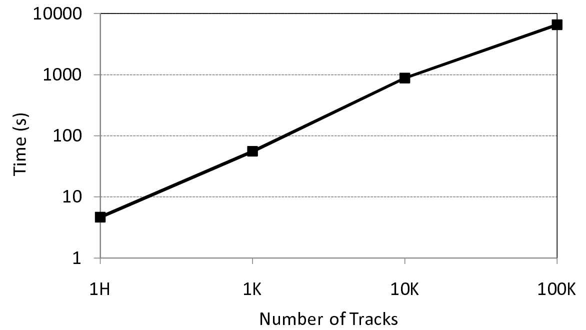


Figure 16: Computational cost of k-medians heuristic

of computing the aggregate similarities is considerable, and as such, pruning trivially bad combinations of representative tracks yields improvements in the performance of the algorithm. We considered one such simple approach: we try to only choose combinations where no two tracks have similarity equal or higher than 0.9. Another approach could be to randomly generate a fixed number of combinations of k tracks, and choose the one in which the k tracks are most dissimilar.

To evaluate the performance of this heuristic clustering algorithm, we chose 10 paths, which were used to generate various numbers of similar tracks. Out of these 10 paths, 5 paths are contained within larger paths (we refer to these as *sub-paths*). If we run the clustering algorithm with $k = 5$, the tracks generated from the *sub-paths* should be placed in the same clusters as the tracks generated from their encompassing paths. For example, we had a path from point A to B, another from point B to C, and a third from point A to C going through B; tracks from A to B and from B to C should be clustered together with those from A to C, which is a super-path of the two previous ones.

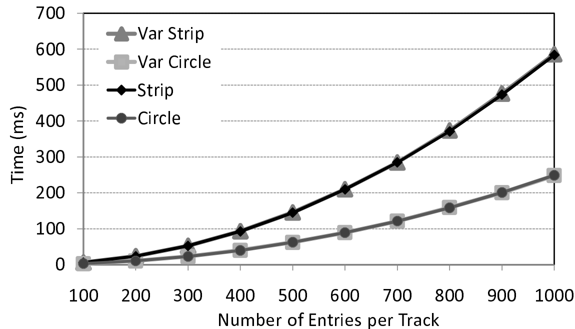


Figure 14: Computational cost of track comparison

We used the aggregate similarity as a metric for quality in our evaluation. Given that all 10 paths used to generate the synthetic tracks were equal or entirely contained within 5 of the paths, we know there exists a clustering assignment for which the aggregate similarity is close or equal to 1.

We ran clustering on 100 tracks (ten tracks generated from each of the 10 paths) and computed the average, as well as the 10th and 90th percentiles of the aggregate similarity across 100 runs of our clustering implementation. Figure 15 shows how the aggregate similarity improves as the number of iterations increases.

Figure 16 shows the computational times (in logarithmic scale) required by the algorithm (stopping after 5 iterations) for clustering 100, 1000, 10,000 and 100,000 tracks. Unlike the naïve approach, for which time would grow exponentially, computational costs for our clustering implementation grow linearly.

Clustering tracks is undoubtedly an expensive operation, which for more than a few dozen tracks would lead to unacceptable response times. When clustering of larger track sets is desirable, it may be performed offline and incrementally, as tracks are inserted into the system. Pre-specified clustering may be performed periodically for a given user, for predefined time intervals, or for particular regions. Our clustering algorithm can also be easily parallelized, which can provide a linear reduction in processing times.

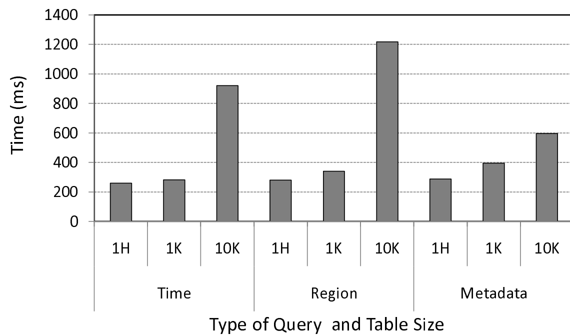


Figure 17: Time taken by time, region and metadata-based queries for various table sizes

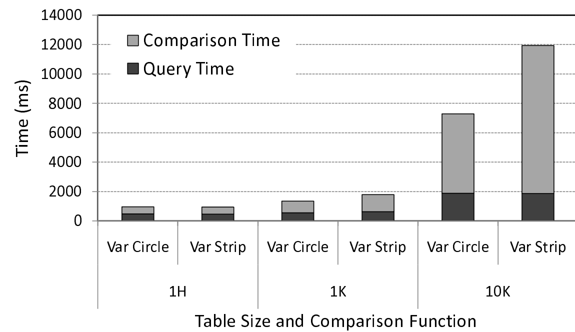


Figure 18: Time taken by *QueryTracksByTrack* for various table sizes

4.3 Querying

We now illustrate the performance of *StarTrack* for various types of track retrieval queries. Our performance benchmarks were performed on a 32-bit Windows Vista PC, with an Intel Core2 Duo CPU 3.00 GHz and 4.00GB of RAM. Prior to each query, the database was restarted, and warmed up by performing a select-all query and an unrelated query of the same type.

Figure 17 shows the time taken for track retrievals done by time, by a rectangular region, and by an XML metadata value for three different table sizes (100, 1000, and 10,000 tracks). The selectivity of each query is approximately 5% of the size of the table. We see that all queries execute in less than 1.3 seconds for even the largest table size we consider. We thus conclude that SQL ServerŠs performance adequately meets our need for fast track retrieval.

Figure 18 shows the performance of the *QueryTracksByTrack* retrieval function. We place a constraint that the resulting tracks must be close to both the start and end points of the target track. Our results provide a breakdown between the time taken to retrieve tracks from the database and that required to perform comparisons. We can see from the figure that retrieval is efficient even when we place constraints. We also see that the performance benefit of circle-based comparison versus strip-based comparison becomes more significant as the number of comparisons grow. Of course, the time required for track comparison grows with the number of tracks that we need to compare against, but in practice, the application writer can prune the set of tracks that need to be compared by using appropriate pruning functions described previously.

5. RELATED WORK

There is an extensive body of work on building platforms for mobile applications, such as Android [1], Windows Mobile .NET Compact Framework [7], iPhone SDK [2] and Symbian OS [6]. These systems primarily focus on providing location (single-point) information on a mobile device. Our work is focused at a slightly higher level of abstraction and enhances the utility of these location service providers by (a) treating tracks as first class objects that can be manipulated in a variety of ways and (b) including services on both the device and the service as part of the same infrastructure enabling large-scale applications to be built easily.

Systems such as Cyberguide [10], the Context Toolkit [8], and the work by Schilt et al. [35] pioneered the use of context-awareness in applications. *StarTrack* can be thought of as an extension of some of these systems in that it includes tracks as part of the context.

CarTel's use of tracks and its grid-based operations on them [22] is similar in spirit to our work. In addition to grids, *StarTrack* allows a wider variety of operations on tracks based on time and user-specified metadata.

As previously mentioned in Section 2.4 a rich set of applications can be built using the facilities in *StarTrack*. Distributed sensor-net applications [21], traffic monitoring [32], route prediction [23, 28, 30, 29], and peer-to-peer collaborative applications [11, 27] are some examples of existing systems that can exploit *StarTrack*.

6. CONCLUSION

A *track*, a time-ordered sequence of locations, is a fundamental new abstraction in support of track-based applications like ride-sharing and location-based collaboration. *StarTrack* is a framework that provides a rich set of operations to ease the development and deployment of track-based applications. The *StarTrack* API helps applications in the recording, uploading, comparing, clustering and retrieval of tracks. Experimental results show that our algorithms are accurate as well as efficient and scalable up to 10,000 tracks.

As part of future work, we plan a full-fledged development and deployment of a number of *StarTrak* applications. We also intend to explore cleaning of location data to eliminate incorrect readings as well as automatically breaking sequence of locations into tracks. This process may also include matching tracks to paths using available blueprints of an area, which could lead to more efficient and accurate posterior operations. We also aim to incorporate new location-based technologies to the *StarTrack* framework, including better use of available network connections, more fault-tolerant services, and possibly more sophisticated comparison and clustering schemes. Finally, privacy in the context of tracks of locations presents challenges that require further exploration.

7. ACKNOWLEDGMENTS

We would like to thank Andrew Goldberg, Moises Goldszmidt, Eric Horvitz, John Krumm, Mark Manasse, Ilya Mironov and Oliver Williams for their advice regarding various technical aspects of this work. We would also like to thank our anonymous reviewers for their comments and suggestions.

8. REFERENCES

[1] Android: An Open Handset Alliance Project. http://code.google.com/android/.

[2] IPhone Dev Center. http://developer.apple.com/iphone/.

[3] Nike + Ipod. http://www.apple.com/ipod/nike/.

[4] SQL Server 2008 Books Online: Spatial Indexing Overview. http://msdn.microsoft.com/en-us/library/bb964712.aspx.

[5] SQL Server 2008 Books Online: Working with Spatial Data. http://msdn.microsoft.com/en-us/library/bb933876.aspx.

[6] Symbian Developer Network. http://developer.symbian.com/main/index.jsp.

[7] Windows Mobile Developer Center. http://msdn.microsoft.com/en-us/windowsmobile/default.aspx.

[8] A. K. Dey and G. D. Abowd. The Context Toolkit: Aiding the Development of Context-Aware Applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[9] A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal of Computing*, 37, 2008.

[10] Abowd, Gregory D. and Atkeson, Christopher G. and Hong, Jason and Long, Sue and Kooper, Rob and Pinkerton, Mike. Cyberguide: A Mobile Context-Aware Tour Guide. In *ACM Wireless Networks*, volume 3, 1997.

[11] Ananthanarayanan, Ganesh and Padmanabhan, Venkata N. and Ravindranath, Lenin and Thekkath, Chandramohan. COMBINE: Leveraging the Power of Wireless Peers through Collaborative Downloading. In *MobiSys*, 2007.

[12] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *INFOCOM*, 2000.

[13] L. Barkhuus, B. Brown, M. Bell, S. Sherwood, M. Hall, and M. Chalmers. From Awareness to Repartee: Sharing Location within Social Groups. In *CHI*, 2008.

[14] N. Bicocchi, G. Castelli, M. Mamei, A. Rosi, and F. Zambonelli. Supporting Location-Aware Services for Mobile Users with the Whereabouts Diary. In *MOBILWARE*, 2007.

[15] K. Chen. On k-Median Clustering in High Dimensions. In *SODA*, 2006.

[16] L. P. Cox, A. Dalton, and V. Marupadi. SmokeScreen: Flexible Privacy Controls for Presence-Sharing. In *MobiSys*, 2007.

[17] D. Pelleg and A. Moore. X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters. In *ICML '00: International Conference on Machine Learning*, 2000.

[18] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[19] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *MobiSys*, 2003.

[20] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual Trip Lines for Distributed Privacy-Preserving Traffic Monitoring. In *MobiSys*, 2008.

[21] Honicky, Richard and Brewer, Eric A. and Paulos, Eric and White, Richard. N-SMARTS: Networked Suite of Mobile Atmospheric Real-Time Sensors. In *Networked System for Developing Regions*, 2008.

[22] Hull, Bret and Bychkovsky, Vladimir and Zhang, Yang and Chen, Kevin and Goraczko, Michel and Miu, Allen and Shih, Eugene and Balakrishnan, Hari and Madden, Samuel. CarTel: A Distributed Mobile Sensor Computing System. In *ACM SenSys*, 2006.

[23] J. Krumm and E. Horvitz. Predestination: Inferring Destinations from Partial Trajectories. In *UbiComp*, 2006.

[24] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):34–45, 1960.

[25] L. Barkhuus et al. Picking Pockets on the Lawn: The Development of Tactics and Strategies in a Mobile Game. In *Ubicomp*, 2005.

[26] A. LaMarca, Y. Chawathe, S. Consolvo, J. Hightower, I. E. Smith, J. Scott, T. Sohn, J. Howard, J. Hughes, F. Potter, J. Tabert, P. Powledge, G. Borriello, and B. N. Schilit. Place Lab: Device Positioning Using Radio Beacons in the Wild. In *Pervasive*, 2005.

[27] L. McNamara, C. Mascolo, and L. Capra. Media Sharing based on Colocation Prediction in Urban Transport. In *MobiCom*, 2008.

[28] N. Marmasse and C. Schmandt. A User-Centered Location Model. In *Personal and Ubiquitous Computing*, 2002.

[29] Navda, Vishnu and Subramanian, Anand Prabhu and Dhanasekaran, Kannan and Timm-Giel, Andreas and Das, Samir. MobiSteer: Using Steerable Beam Directional Antenna for Vehicular Network Access. In *MobiSys*, 2007.

[30] A. J. Nicholson and B. D. Noble. BreadCrumbs: Forecasting Mobile Connectivity. In *MobiCom*, 2008.

[31] V. Otsason, A. Varshavsky, A. LaMarca, and E. de Lara. Accurate GSM Indoor Localization. In *Ubicomp*, 2005.

[32] P. Mohan et al. Nericell: Rich Monitoring of Road and Traffic Conditions using Mobile Smartphones. In *ACM SenSys*, 2008.

[33] R. O. Duda and P. E. Hart. Pattern Classification and Scene Analysis. In *New York: John Wiley and Sons*, 1973.

[34] R. Spanek et al. The BlueGame Project: Ad-hoc Multiplayer Mobile Game with Social Dimension. In *CoNEXT*, 2007.

[35] W.N. Schilit. *System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, 1995.