dbTRACE: A Scalable Platform for Tracking Information

Filipe Guerreiro
filipe.m.guerreiro@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2016

Abstract

dbTRACE is used to store city road map information, user tracking information and provide tracked data for analysis. This platform is designed to address the prevalent issues of the TRACE initiative. The idea of TRACE was to reinforce the importance of a more active lifestyle to improve health and city environments. dbTRACE uses TitanDB, a graph database engine that uses Cassandra as a storage back-end and communicates with an ElasticSearch cluster as a geospatial index to distribute data across a cluster of machines in order to provide highly scalable storage and traversal performance. User trajectories are processed using Barefoot, a Map-matching solution that matches points to a road network obtained from OpenStreetMap. Barefoot can be scaled out using the Spark framework for distributed processing. The trajectories are then inserted into TitanDB which can then be quickly traversed and queried for statistical analysis and reward validation.
Keywords: Graph Database, Map-matching, Distributed, Tracking, Trajectory

1. Introduction

Advancements in sensor technology have made it possible to generate large amounts of geospatial data. In turn, this new reality has made large-scale tracking initiatives more interesting. Physical inactivity has become an import public health issue. There is a growing interest in influencing better physical activity behaviors. As a consequence, this leads to less pollution, less traffic, better environment and quality of life improvements. One of the initiatives that has originated from these goals is called TRACE. The TRACE initiative is an European project being managed by INESC-ID with the goal of getting more people walking and cycling. Participants are incentivized by the use of rewards, such as prizes and discounts.

These large amounts of data need a storage database capable of storing and querying this kind of information efficiently. Storage database technology has, since the 1970's [13, pp. 5–8], had one *de facto* solution, the relational database. However, relational databases have a few problems which prevent them from scaling well. The first is the requirement of fixed table schema, which leads to inefficient use of space when you have similar objects with one or two different attributes, leading to several null values, or additional tables which store the differences, leading to additional table join operations.

Furthermore, as the amount of data increases and the limits of a single machine is reached, there is a need to distribute the service between several machines. In this environment, the use of Atomicity, Consistency, Isolation, Durability (ACID) transactions becomes a bottle-neck, requiring expensive two-phase commit protocols. In addition, join operations become very expensive when the tables are partitioned between several machines.

Ten years ago, Google and Amazon [3, 5] started the (noSQL) movement of simpler, scalable databases designed for very big amounts of data (i.e. Big Data). This movement has originated many different solutions, based on different data models. Popular taxonomy introduced by Rick Cattell [2] distinguishes the different databases into Key-value, Document, Extensible-Record and Graph. Graphs, in particular, provide a natural approach to model relationships between objects. The use of a graph-based data model makes it possible to store and query data in a way that semantically represents its own structure. Just as it can naturally represent networks of friends or links between web pages, it can also naturally represent networks of roads, points of interest, shops, hotels, cyclists and their trajectories. Furthermore, graph theory is a well studied field in both Computer Science and Mathematics, with many efficient and optimized algorithms already developed.

2. Related Work

2.1. Property graph

A graph is a mathematical structure used to model relations between objects. A graph is made up of a

set of objects where some pairs of objects are connected by links. These objects are called vertices, while the links that connect these vertices are called edges.

The most common graph implementation present in most graph databases (TitanDB included), is the Property graph. The Property graph allows the representation of labeled vertices, labeled edges, and attribute data (or: properties) for both vertices and edges.

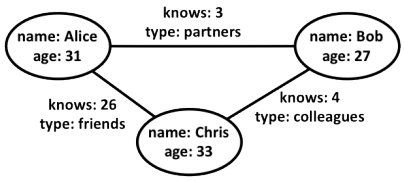

Figure 1: An example of a property graph (Groovy-Mag 2013)

A simple example can be seen in Fig.1, with three vertices representing people and edges representing the relationship between them. All vertices have three properties, a name property and an age property, as well as a unique id property which is not shown. The edges have two properties, a label with the semantic meaning of the edge and its unique id, as well as a type property describing the relationship between people.

2.2. Geospatial Information

Geospatial information is information about a physical object that can be represented by numerical values in a geographic coordinate system. More specifically, it represents the location, size and shape of an object on a planet, such as a mountain, a building, a road or a town.

Transportation and mobility networks, such as roads, aqueducts and railways, are conceptually represented by spatial networks. Spatial networks are graphs whose vertices and edges are located in a space equipped with a metric, such as the Euclidean distance[1] [1].

An urban spatial network can be modeled with the vertices representing intersections and junctions, while the edges represent streets and roads the individuals can take between locations [7].

2.3. Map Matching

Map matching is the process of converting a sequence of latitude/longitude points into a sequence of road segments.

Data received from sensors often has errors that need to be corrected. The size of a trajectory can often be a problem due to high sampling rate of devices, leading to overuse of storage space and processing time. Furthermore, it may be difficult to distinguish which road certain sequences of points belong to, due to overpasses and crossings.

Map-matching is a well-researched problem, with many implementations already available. Examples of open-source implementations include OSRM (`http://project-osrm.org/`), GraphHopper (`https://graphhopper.com/`) and Barefoot (`https://github.com/bmwcarit/barefoot/`) [11].

GraphHopper uses a variant of Marchal's algorithm [10], while both OSRM and Barefoot use the HMM model [12], which is the current state-of-the art. OSRM is used by MapBox, while Barefoot is used by BMW. Both use OSM data for building an in-memory map graph. Out of the box, OSRM provides pre-made configurations for different profiles, such as pedestrian, bicycle and car. Barefoot only offers a profile for cars, but is capable of using Apache Spark for distributed map matching computation.

We opted for Barefoot because we can fix its limitation by creating our own pedestrian and bicycle profile and a few configuration adjustments.

2.4. Stay-Point Detection

Stay points are points that denote locations where the user has stayed for a while, such as shopping malls and tourist attractions. There are two types of stay points. The first is the single point stay-point, where the individual remains stationary for over a time threshold. This tends to occur when the individual enters a building and loses the satellite signal until returning to the outdoors. The second is the most common, where users walk around within a small spatial region for over a time threshold. This tends to occur when the individual wanders around some places, like a park or a campus [15].

These stay points can turn a trajectory as a series of time-stamped spatial points to a series of meaningful places. This in turn, is used for a variety of applications such as travel recommendation, taxi recommendation, destination prediction and method of transport used.

2.5. TitanDB

TitanDB is an open-source graph database developed in Java and licensed under Apache 2. It was developed by Aurelius and first released in 2011, recently being acquired by DataStax in 2015.

As can be seen in Fig. 2 An interesting characteristic of Titan is its need to plugin with different technologies, including storage back-ends such as Cassandra (`http://cassandra.apache.org/`) and HBase (`https://hbase.apache.org/`), search engines such as Lucene (`https://lucene.apache.`

[1]The Euclidean distance or Euclidean metric is the straight-line distance between two points in space.

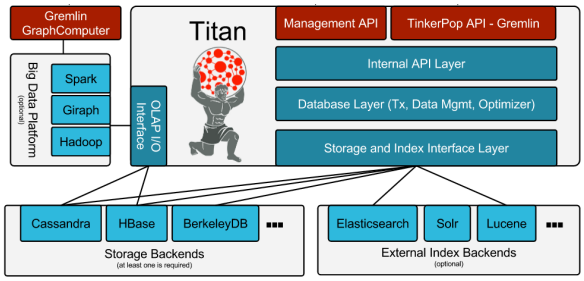

Figure 2: A high-level Titan architecture view (TitanDB 2013)

org/) and ElasticSearch (https://www.elastic.co/) for speeding up complex queries, and even provide OLAP[2] operations with Hadoop and Spark[3].

TitanDB's distribution capabilities will depend on the chosen storage back-end. HBase is a CP model, giving preference to consistency at the expense of availability. Cassandra falls in the AP model, giving preference to availability at the expense of consistency (the completeness of the answer to the query), Cassandra has a master-less ring architecture, where scalability in both read and write form grows with the number of nodes.
Geospatial capabilities can be obtained with the use of ElasticSearch. This open-source search and analytics engine provides two spatial indexes, a Geohash and a Quadtree. Spatial data type support includes points and a general shape type, which can define lines and polygons. As for topological function support, it supports the Within and Intersects functions. It also provides support for the Distance, Range and BoundingBox spatial distance functions.

3. Architecture

3.1. dbTRACE

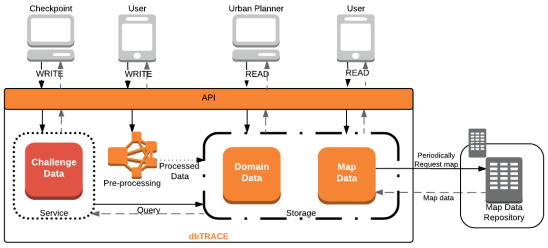

Figure 3: Global view of dbTRACE and the interactions between its main components.

dbTRACE handles the storage information of three different types of users. Local businesses (**Checkpoints**) are interested in attracting new clients who arrive by bicycle or on foot (**Users**), which in turn, are interested in obtaining discounts and rewards. Furthermore, cities are interested in using the obtained tracking information to improve urban planning (**Urban planners**/**Analysts**).

An overview of dbTRACE can be seen in Figure 3. The storage component is divided into two parts, the **Domain Data Module** and the **Map Data Module**. The **Domain Data Module** stores user information and trajectory identifiers, as well as checkpoint information. The **Map Data Module** stores the road network of the city, as well as the processed trajectories as a sequence of roads. The road network information is also periodically updated through an external map repository to obtain changes about new or closed roads.

Applications and services communicate and access the stored data through an **API** which defines the operations each one is allowed.

There is also a **Pre-processing** component that takes new trajectories sent by the users and processes them in order to correct and compress the data, extract new semantic value, as well as applying fraud countermeasures. More concretely, there are 2 processes involved. First, there is the **Stay-point detection** process to extract semantic value, namely, the places where the user stopped and stayed for a while, which is useful for determining which Checkpoints the user might have visited. The second is the **Map-matching** process which corrects and inserts the trajectory into the Map Data module, determining the roads and streets in the user's itinerary.

Finally, there is a **Challenge service** developed by colleague Antnio Pinto as part of his master's dissertation, called *Promoting urban bicycle use through next-generation cycle track-and-score systems* [4]. Its role is to store and process **challenges**, which are a custom list of criteria that the users need to meet to earn a reward. This criteria can range from a specific time period, geographic area, number of visits or luck. After processing a challenge, this service queries the storage module to determine eligible users. To support this service, we implemented the **Reward queries**, described further in Section 4.2.

3.2. Domain Data

The **Domain Data** module stores the users movements, their personal details, as well as checkpoint information and reward points obtained. These reward points are earned when a user walks by a checkpoint. Figure 4 shows a logical graph model representing all of the TRACE entities and their relationships.

Checkpoints represent the shops or local busi-

[2]OLAP stands for OnLine Analytical Processing. It consists of multidimensional analysis of business data, providing the capability for complex calculations, trend analysis, and data modeling.

[3]Spark is similar to Hadoop, but allows for data to persist across nodes for repeated querying and data mining operations, http://spark.apache.org/

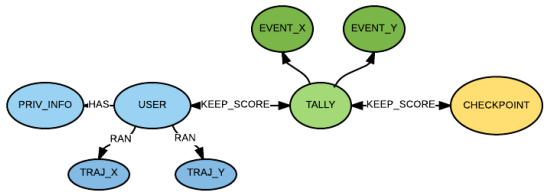

Figure 4: Logical Domain Data Model in Graph format.

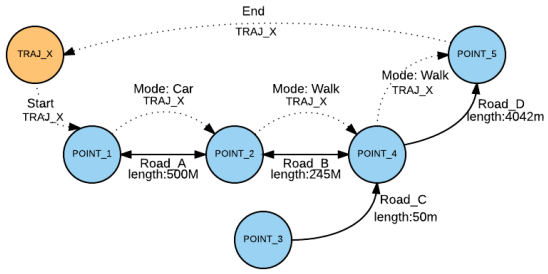

Figure 5: A graph representation of a road network with a processed trajectory example. The blue nodes show geographical points in the road network, representing intersections and junctions. The orange node represents a user trajectory. The edges with full lines represent different road segments, while the edges with dotted lines represent the orange trajectory's travel sequence.

nesses. They are uniquely identified by a **name**. They also have a geographic **location**, represented as a longitude and latitude pair.

Users represent the customers who run or cycle to shops. They are the moving entities of the database. They have a **unique ID** that is used for referring to a User inside and outside the database (they can be viewed by Checkpoints for delivering rewards). **Private information** such as *e-mail* for being contacted automatically when a prize is earned, or the *fiscal number* for cash-prize eligibility. This information is facultative. A **list of trajectories**. Each trajectory here is simply represented by a unique identifier to the trajectory stored in the Map Data Module. They are preserved for one month or until being verified for reward-eligibility challenges. This is done in order to minimize exposure to data that can be used to trace back to the identity of the user. And finally, a **reward point tally** (shown as "Tally" in Figure 4) for each different checkpoint. Each time a user passes a checkpoint, an event is stored, recording the time it happened. Reward events are kept for Path Inference operations (for fraud detection), as well as to allow shop owners to reward users, that, for instance, showed up on a particular day, or to users that showed up over 80% of the days of the month. Lifetime points are kept so that store owners can reward loyal users that attend over a certain milestone-number of times for example. Afterwards the points are added to the lifetime points tally.

3.3. Map Data

The model representing the road network and user trajectories can be seen in Fig. 5.

Nodes represent the intersections and junctions between streets. Each node is uniquely identified by the map provider's identifier. They must also have a **position** property, representing a single point consisting of a latitude and a longitude pair.

Edges connecting the two nodes represent the streets between those nodes. Each edge is also uniquely identified by the map provider's identifier. In addition, each edge has 3 properties. The **name of the street** to which it belongs to, the **length** of the street segment as well as the **type** of street. The **type** property can have the following values: motorway, road and residential.[4] A motorway is a path where only motorized vehicles may use. A road is a path where both motorized vehicles and non-motorized vehicles, as well as pedestrians may use. While a residential way is a path where only pedestrians and non-motorized vehicles may use. Optionally, it may have an **inclination** property (this property is often missing from map providers).

User trajectories are stored within the road network. A trajectory starts with a node, called the **trajectory node** (seen as orange in Fig. 5), which contains 2 properties. It has a unique **identifier** and the total **length** of the trajectory. The trajectory node has a **start edge**, which has the **start time** of the trajectory and points to the first node in the road network. The trajectory path is represented as edges between nodes in the road network, which we call **path edges**. Each path edge carries the trajectory's unique identifier, the **start time** of the user entering the road segment[5], as well as the transportation mode used. Finally, the trajectory ends with an **end edge** connecting the last road network node to the respective trajectory node.

3.4. Stay-Point Detection

Before a trajectory is inserted into the Map graph, we start by trying to identify the **Stay-Points**. This is done in order to determine where the user stopped for some time, which is useful for identifying which stores (checkpoints) the user may have entered.

To determine stay-points, we use the algorithm

[4]Our classification is a simplified version of the Open Street Map road tag system, `http://wiki.openstreetmap.org/wiki/Key:highway`

[5]The length of stay between points can be calculated by subtracting the start time of the subsequent path edge and the current edge

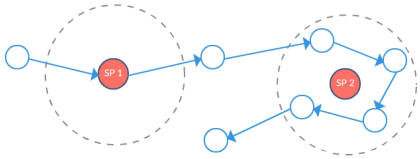

Figure 6: Stay point 1 shows a case when the user remains stationary for a time period exceeding a threshold. In stay-point 2 the user wanders around within a certain spatial region for a period.

proposed by Li et al.[8]. Their algorithm starts by first considering a point, looking at its successor and seeing if it is within a space and time distance threshold. If it is, then it calculates the mean of those points and uses that as the reference point. Then, it tries to add the next point and do the same check. This continues until no more points are within the space and time thresholds, obtaining a set of points that belong to the same "stay point". This process then repeats for the rest of the trajectory.

3.5. Map-matching

After removing the trajectory's stay-points, we use a **Map-matching** algorithm to convert the trajectory's points into a series of matching road segments. To do this, we use the algorithm introduced by Newson and Krumm [12] which works well in noisy and low sampling settings and is implemented by many routing services today [9, 11].

In their model, the states of the HMM are the road segments and the state measurements are the tracked location measurements. Road segments nearer to the point are assigned a higher probability. Then, the algorithm computes the transition probability for every pair of adjacent hidden states in the chain such that the probability of the next state is dependent only on the previous. When the last point is processed, it finds the maximum likelihood path over the Markov chain that has the highest joint emission and transmission probabilities.

The Map-matching algorithm takes a trajectory z with points z_1, z_2, ..., z_n. For each point z_t, the algorithm fetches the set of most likely candidates S_t from the map, based on the euclidean distance between z_t and s_t (called filter probability). For each iteration where $t > 0$, the most likely sequence between $s_t - 1$ and s_t is calculated based on the routing distance between them (called sequence probability).

4. Implementation

Figure 8 shows the different interacting components of the dbTRACE implementation. There are three essential components to the system. Applications and services communicate with the system through

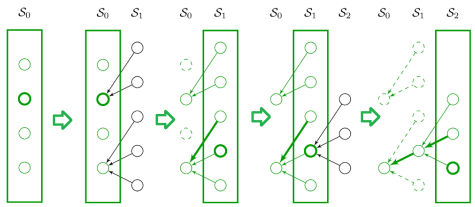

Figure 7: On each matching iteration, the algorithm fetches candidate set $S_t - 1$, calculates for each candidate s_t filter and sequence probability and its most likely predecessor in $S_t - 1$.

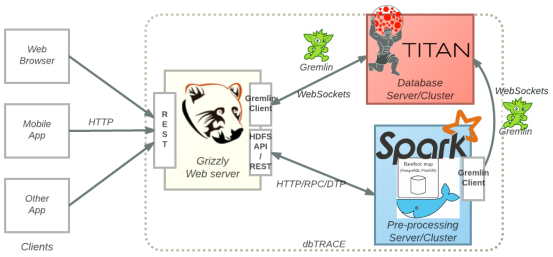

Figure 8: A high-level view of the interaction between the different components in the dbTRACE system.

the **Web Server** (using Grizzly), which provides an HTTP REST API to the system. These API calls are then transformed into calls to the internal components of dbTRACE.

To store, retrieve and manage user information and trajectory data, we use a **Database Server/-Cluster** that uses TitanDB, Cassandra and ElasticSearch.

Before storing new trajectories, the information is sent to the **Pre-processing Server** (Barefoot) / **Cluster** (Barefoot and Spark) that does Map-matching and Stay-point detection. Upon finishing the pre-processing steps, the stay-points are used to find the nearby checkpoint and add to their reward tally, and the map-matched trajectory is inserted into the database.

4.1. Database

The Database component uses the TitanDB graph engine for graph-based data modeling and query language support. Scaling-out is accomplished using the Cassandra storage back-end, as well as the ElasticSearch index back-end to enable distributed geospatial query support. Figure 9 shows how these components interact.

4.2. Database Queries

Queries to TitanDB are made using Gremlin. Gremlin is a graph-traversal extension that is incorporated into existing languages.

Queries to dbTRACE can be divided into two categories, the **Reward queries** and the **Analy-**

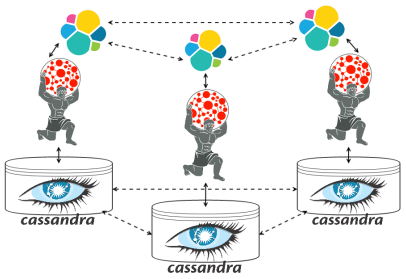

Figure 9: A high-level view of the distributed architecture using Cassandra as a persistent data storage, ElasticSearch as an index for speeding up geospatial queries, and Titan as a transaction and data manager.

sis queries. The Reward queries are requested by Checkpoint owners that wish to obtain the list of users that are eligible for receiving a reward. They involve determining the distance a user traveled in total, in a specific area or in a specific route. The Analysis queries are issued by entities interested in statistics of the data set, such as urban planners. They include calculating how many users used a given road, how many users traveled between two different regions, or the average distance between users that traveled between two regions.

Below, we describe the *Total Distance traveled in a specific route* query and the *Average Trip Distance between origin-destination query*.

Total distance traveled in a specific route

The goal of this query is to calculate the total distance traveled by a particular user, represented by the `sessions` identifiers, over a defined `route`.

This query is split into two functions. The first function (*distanceInRoute()*) first searches the graph for the beginning vertex of each trajectory. Then it traverses each trajectory, obtains the list of traversed vertexes and calls the compareRoute() (*map()* step) function for each, returning true if it is a match or false otherwise.

The function (*compareRoute()*) compares the user trajectory to a given route. Comparing different trajectories is a well-studied problem [14], with research that tries to tackle the inherent imprecision of the tracking devices. In our case however, we have dealt with these imprecisions during the pre-processing stage. This means that we have reduced the problem from comparing two sequences of geospatial points to the more general problem of comparing two sequences. As a consequence, we can use a sequence comparison technique such as the the Levenshtein's distance or the Longest Common Subsequence (LCS).

In our route comparison function (*compareRoute()*), we say there is a match if the user route and target route have a subsequence

Listing 1: Total distance traveled in specific route query

```
distanceInRoute = { def sessions, route ->
  userSessions = g.withSack("").V()
    .has('sessionID', within(sessions)).outE('session').outV();
  listComparisons = userSessions
    .sack{m,v -> v.value('sessionID')}
    .repeat(outE().filter(filterEdgeBySession).inV())
    .until(hasLabel('session')).path()
    .by("location").by(constant('edge'))
    .map({compareRoute(it, route)});
  return listComparisons;
}

compareRoute = { def userRoute, def route ->
  trajectory = userRoute.get().objects().grep{it != "edge"};
  MIN_MATCH  = route.size() * 0.75;
  start = -1; end  = -1;
  /* is user doing the route in the inverse direction? */
  def startNode = route[0];
  def endNode = route[route.size()-1];
  for(i = 0; i < pointList.size(); i++) {
    if(trajectory[i].equals( startNode ) && start == -1)
      start = i;
    if(trajectory[i].equals( endNode ) && end  == -1)
      end = i;
  };
  def result = false;
  if(end < start && end != -1 && start != -1) {
    match = lcs_length(trajectory, route.reverse());
    result = (match > MIN_MATCH)? false : true;
  }
  else {
    match  = lcs_length(trajectory, route);
    result = (match > MIN_MATCH)? false : true;
    if(result == false) {
      match = lcs_length(trajectory, route.reverse());
      result = (match > MIN_MATCH)? false : true;
    }
  }
  return result;
};
```

with at least 75% of the size of the target route (to provide some lee-way to the user). To compare two trajectories (the route and the user trajectory), we use the LCS algorithm developed by Hirschberg [6]. This algorithm has asymptotic time complexity $O(mn)$ and space $O(min(n, m))$ where m and n are the sizes of each sequence.

Before applying the LCS algorithm, we need to take into account the cases where the user travels the route in the opposite direction. To do this, first, we check if the target route's `start` and `end` points are present in the user route and what their order is, i.e., if `start` comes after `end` or vice-versa. If so, we reverse the route before applying the LCS algorithm. If either the route's `start` and `end` nodes do not appear in the user route, we cannot tell which direction the user is going. In this case, we start by applying the LCS algorithm and verify the result. If there was not a match, we test for the reverse direction.

Average trip distance between origin-destination

This query gets all the trajectories between area `originArea` and `destnArea` and calculates the average distance.

Listing 2: Average trip distance query

```
avgTripDist = { def originAr, destnAr ->
  filteredSessions = g.withSack("").V()
    .has('location', geoWithin(originAr))
    .inE('session').has('type', 'start').outV()
    .inE('session').has('type', 'finish').outV()
    .has('location', geoWithin(destnAr)).outE('session')
    .has('type', 'finish').inV();
  return filteredSessions.sack{m,v -> v.value('sessionID')}
    .repeat(outE().filter(filterEdgeBySession).inV())
    .until(hasLabel('session')).path()
    .by('location').by(constant('edge'))
    .map(calculateTripDist).map({it.get().get()}).mean();
  }

calculateTripDist = { def points ->
  def a = Optional.empty();
    /*removes path history from the initial filter phase*/
  def pointList = points.get().objects().drop(6);
  def sessV = pointList.first();
  for(def b in pointList.grep{it != "edge" && it != sessV}){
      if (a.isPresent()) {
          def (d, p) = a.get();
          d += p.getPoint().distance(b.getPoint()); /*km*/
          a = Optional.of([d, b]);
      } else a = Optional.of([0, b]);
  };
    return a.isPresent() ? Optional.of(a.get()[0]*1000): a;
}
```

Determining the average distance is divided into two phases. First, we have the filter phase which corresponds to the `filteredSessions` attribution in function *avgTripTime()*. To calculate this, we do a series of filter operations. First, we do a spatial index look-up to get all the points that are in the `originAr`. Then we check if they are the first point in the trajectory (by walking back and see if there is a session node). Then, walk backwards again to get the last point in the trajectory (edges in Titan are bi-directional, creating a loop), to apply a final spatial *contains* query to check if it is in `destnAr`.

Now that we found the list of matching trajectories, the algorithm traverses each of them from start point to end point (*repeat.(...).until(...)*) and stores the path history (*path()*). We then modify the path output to show the *location* property of the vertices (*by('location')*) and pass it (*map()*) to our function *calculateTripTime()* to calculate the distance of the trajectory.

4.3. Map-matching

Map-matching is done using Barefoot, an Open-source Java library and a map server that provides access to street map data from OpenStreetMap using a state-of-the-art Hidden Markov Model algorithm. By default, Barefoot provides map-matching only to motorized vehicles. To adapt it to map-match pedestrian and cycling we need to modify two files. The first is the configuration file (`bfmap/road-types.json`) which identifies which road tags (*motorway*, *trunk*, *primary*, ...) are imported from OSM. Here we add roads which are typically prohibited to cars and bikes such as *residential*, *footway* and *cycleway*. Next, we remove one-way and turn restrictions which pedestrians often do not take into account. We modify the `com.bmwcarit.barefoot.road.BaseRoad` class to always ignore those attributes (setting the flags to false).

Next, we adjusted the HMM matching algorithm parameters in file `config/server.properties`, as shown in Listing 3.

Listing 3: Map matching server configuration settings

```
matcherSigma=15 #Urban GPS error radius
matcherMaxDistance=1500 #Max road distance between pts
matcherMaxRadius=35 #Distance to consider candidate roads
matcherMinInterval=5000 #Skip high-rate measurements
matcherMinDistance=10 #Skip close-by points
```

5. Results

5.1. Database Query Analysis

This Section measures the run-time of the different implemented queries and analyzes their operations to determine their respective bottlenecks. We performed this test on a machine using an Intel Core i5-4200U 1.9 GHz processor with 4GB of RAM and 128 GB SSD storage.

We used BerkeleyDB for the storage database with a dataset of 180 000 vertices (approximate number of vertices in Lisbon). There were 100 trajectories (expected average number of filtered trajectories measured in a query), with an average size of 547 edges (expected average size of 30 minute trajectory run).

The run-time was measured using the `profile().cap(TraversalMetrics.METRICS_KEY)` operation. This allows us to see each traversal step run-time, at the expense of overhead on the final run-time (5 to 10 times greater).

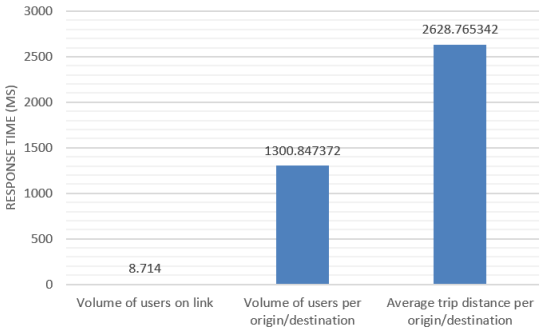

Figure 10: A comparison of the response time in milliseconds (y axis) between the different Urban Planner queries (x axis).

Fig. 10 shows how the response time varies with query complexity. Calculating the number of users that passed through a certain road is very quick, as expected. About 60% of the run-time is a graph traversal from the beginning to the end of the road, while 30% is used on the deduplication of repeat users and the final 10% is for the count-step.

Counting the number of users that started a trip in `origin area` and ended the trip in `destination area`, there is a sharp increase in response time. There is first a spatial filter operation (ES index look-up) that accounts for 5% of run-time. Each node has one edge traversal and a filter operation (10%). For each filtered node, there is one spatial filter ES look-up, representing 85% of the run-time, which increases further as the filtering area increases and more nodes need ES look-ups. This can likely be improved if we only do two spacial look-ups for finding the origin and destiny sessions, and then finding the common elements in both sets using an algorithm such as the merge part of the *merge-sort* algorithm.

Calculating the average distance of trips that start in `origin area` and end in `destination area` is a good example of an expensive OLAP query, composed of a mix of the previous queries. First it has a spatial filter operation to obtain the list of matching trajectories and then it has a traversal loop for each of these trajectories (each will traverse 547 edges on average). For this data-set, the spatial index look-up occupied about 30% of the run-time, while the traversal about 70%. This query can be easily optimized by adding the trajectory distance to the session node as a property. This would eliminate the traversal operation of the query, requiring just the index look-up.

5.2. Distributed Database Performance

Titan has been shown to serve thousands of OLTP requests per second.[6] However, many of the queries we use are OLAP-style analysis queries.

To test the performance of the database when used in a distributed fashion, we tested the system response time to OLAP queries as a function of the number of nodes, as well as a function of the number of concurrent queries. We used Amazon Web Services (AWS) (`http://aws.amazon.com/`) to deploy a small Elastic Compute Cloud (EC2)[7] cluster.

For the instance type, we chose `m4.large` instances. These instances have 2 vCPUs with 2.4 GHz Intel Xeon E5-2676 processor and 8 GB memory. We ran Cassandra and Gremlin-server on each of these machines that came bundled with Titan 1.0. Each machine was also running an ElasticSearch (v1.5.2) node. Furthermore, we kept Cassandra's consistency level at one and the replication-factor for both Cassandra and ES equal to the number of nodes until 3. Further nodes kept the replication-factor of 3.

The dataset was composed of 200 000 points (Lisbon has about 180 000 vertices) and 100 trajectories, each with a length of 1000 edges (expected average trajectory size for a 1 hour run). All results were measured using the `profile().cap(TraversalMetrics.METRICS_KEY)` step operation.

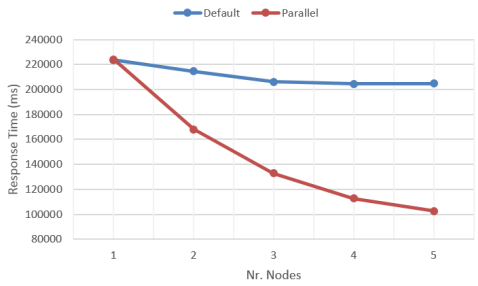

Figure 11: Response time in milliseconds as a function of the number of nodes in the cluster. The Default line shows the results of the test on the Average Trip distance query over a result set of 100 trajectories. The Parallel line shows a modified client that distributes the requested trajectories evenly between the nodes.

Figure 11 shows the relative performance of the system as it goes from a single machine to an increasingly distributed cluster of machines. The Default line shows that the performance of the query does not improve significantly as more nodes are added to the system (the improvement likely comes from the added ElasticSearch nodes concurrently doing spatial searches). This means that TitanDB by default runs all deep search traversals (each trajectory) in the same node. By modifying the query to explicitly send each piece of the query processing load to a different node in the cluster (Parallel line), we see a logarithmic improvement in runtime. Increasing the cluster from 1 node to 2 improves the response time by 26%, while adding the 4th and 5th node improve only by 8%. We suspect that the added data partitioning and communication overhead place a hard limit on scalability on these (OLAP) types of queries.

The greatest benefit of adding more nodes comes in system throughput. Figure 12 shows the effect of the number of nodes as the number of concurrent users (using the distribution-aware client) on the system increases.

As more users are added to the system, the greater the improvement of the distributed setting,

[6] Titan scaling, `https://dzone.com/articles/titan-provides-real-time-big`

[7] EC2 is the AWS service that allows launching virtual machines in one of Amazon's data-centers.

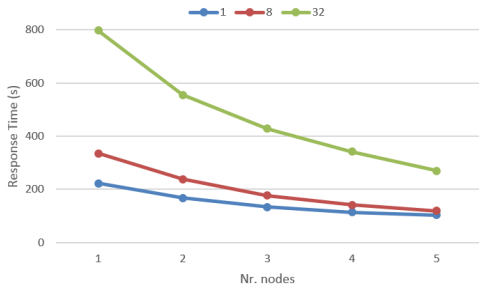

Figure 12: The number of concurrent clients (1, 8 and 32) as a function of the number of nodes in the cluster, using the parallel Average Trip Distance query. For both the 8 and 32 series, the queries were sent at the same time and the time was measured when all results were returned.

likely compounded by the caching mechanisms used by both Titan and ES. We can see that with 8 clients, we get gains of, on average, 17% per added node. In an environment with 32 clients, we see gains of on average 24% per added node.

5.3. Map-matching

These tests evaluate the performance of the Map matching module in terms of: accuracy, running time and distributed running time. The configurations of were described in Section 4.3. Map data was for a region of Portugal encompassing Lisbon, Sintra and Cascais, consisting of 236 961 ways.

Accuracy

Table 1 shows the accuracy of the Barefoot Map-matching implementation for 3 different trajectories. The first is a bicycle trip with 3081 points. The second and third trajectories are pedestrian trips taken in Lisbon. There is a running trip, with 1976 points, as well as a walking trip, with a total of 2205 points.

Measuring the accuracy of a map matching algorithm requires comparing the resulting track with the *calibrated* track. The calibrated track refers to the optimal result we expected to see from the map matching procedure. To obtain this data, we applied the map matching algorithm to the measured track and displayed the result graphically (using ArcGIS Editor) so we could hand-match the cases where the algorithm has made an error. To measure the error between the resulting track and the calibrated track, we calculate the length of segments that were incorrect. That is, we sum the length of incorrect segments that were added or removed, divided by the total length of the calibrated track.

In our tests, we got an average (on our small sample size) of 90% map-matching accuracy rate. Map-matching performed better on the bicycle route than the other routes. We attribute this to the tendency of pedestrians to take more shortcuts,

Table 1: Comparing the accuracy for the three different tested routes. For each case, we indicate the length of the calibrated trajectory, followed by the length of the incorrect segments of the resulting track and the overall accuracy percentage according to the formula mentioned above.

	Calibrated trajectory	Incorrect segments	Correct match
Bicycle route	40462 m	1561 m	97 %
Running route	19326 m	3958 m	83 %
Walking route	22763 m	2529 m	90 %

through which there is no corresponding road in the map network. Another factor to consider is that the tracks taken were a few years old. The state of the OSM map data is more likely to be out-of-date in a city environment that changes frequently, compared to the rural environment of Sintra.

Performance

We tested the performance of the Barefoot algorithm, running on an Intel Core i3 CPU @ 3.07GHz and 4 GB of memory with Ubuntu 14.04. Barefoot was configured in a Docker container[8] backed by a PostgreSQL/PostGIS database.

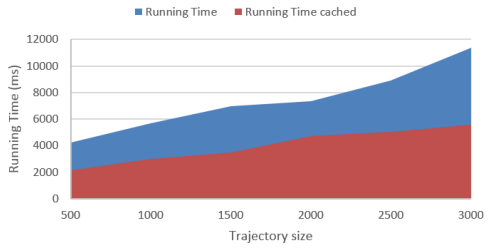

Figure 13: This represents the map-matching running time in milliseconds as a function of the number of points in the trajectory. The blue area shows the running time when there is no cached data. The red area shows the running time when Postgres has already cached the search area.

Figure 13 shows the performance of the algorithm as a function of the size of the trajectory. This relationship is strongly linear. The maximum value we obtain, for a trajectory with 3000 points, is a running time of approximately 11 seconds. This running time is improved by the caching mechanism used by PostgreSQL, cutting the run-time in half.

[8]Docker is an Operating-system-level virtualization software. Programs in virtual partitions (typically called containers) use the operating system's normal system call interface and resources instead of being emulated by an intermediate virtual machine. Like VM images, container images are a snapshot of a system state that can be later started and run. Website, `https://www.docker.com`

In a synchronous system, this run-time of several seconds for one request makes it difficult to scale to thousands of users. For our use case, which does not require map-matching in real time, it is a good solution. However, Barefoot also supports a distributed configuration using Spark, which we can use to try to improve these results.

Distributed Performance

We tested the performance of the distributed Barefoot environment using a small AWS EC2 cluster, with a `m2.medium` instance type, which has 2 vCPUS and 4 RAM backed by an SSD storage. The tested track was the Sintra bicycle route.

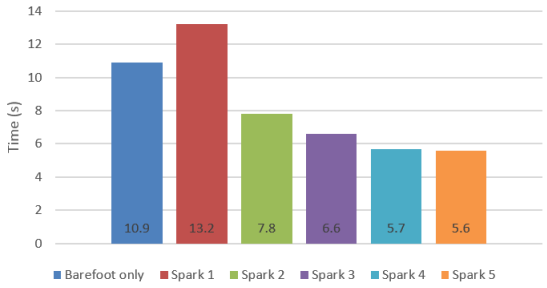

Figure 14: This chart provides a relative comparison of the running time in seconds in different computation set-ups. The first bar on the left serves as a reference point, running the usual barefoot application on one machine. The following bars are running our Spark application, with the number reference indicating how many machines are in the cluster.

The performance gains for each added node decrease with the number of nodes added. From the base setup running a single barefoot server, we see a 31.9149% improvement in running time when going to a 2 machine setup with Spark, a 16.(6)% drop from 2 to 3, 14.6341% from 3 to 4, and then it stabilizes, showing no significant gains. At 4 machines there is a 49.1429% improvement over the original (non-Spark) case.

This shows that using the distributed Spark configuration reduces run-time up-to half, and is thus better for synchronous systems where response time is prioritized. However, it is less efficient than the single-server set-up from a system throughput standpoint. Consider four different trajectories with the same size as the Sintra track and four machines running a separate Barefoot server. The 4 tracks will finish in 11 seconds. The distributed setting will finish each track in 5.7 seconds for a total of 22.8 seconds.

6. Conclusion

There is a scarcity of platforms for large-scale managing and storing of user tracking data. This document presents dbTRACE, a scalable system composed of a distributed graph database using TitanDB for storing and querying large amounts of user trajectories, and a pre-processing cluster using Barefoot and Spark for stay-point detection and map-matching.

In the future, it would be interesting to implement the map-matching algorithm using Gremlin and running it in Titan instead of relying on Barefoot (which uses a relational database to store map data).

References

[1] M. Barthelemy. Spatial networks. *Physics Reports 499:1-101*, 2011.

[2] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[4] A. M. de Sousa Nunes Marques Pinto. Promoting urban bicycle use through next-generatio cycle track-and-score systems. Master's thesis.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[6] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[7] V. Kanjilal and M. Schneider. Spatial network modeling for databases. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 827–832. ACM, 2011.

[8] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma. Mining user similarity based on location history. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, page 34. ACM, 2008.

[9] D. Luxen and C. Vetter. Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 513–516, New York, NY, USA, 2011. ACM.

[10] F. Marchal, J. Hackney, and K. Axhausen. Efficient map matching of large global positioning system data sets: Tests on speed-monitoring experiment in Zürich. *Transportation Research Record: Journal of the Transportation Research Board*, (1935):93–100, 2005.

[11] S. Mattheis, K. K. Al-Zahid, B. Engelmann, A. Hildisch, S. Holder, O. Lazarevych, D. Mohr, F. Sedlmeier, and R. Zinck. Putting the car on the map: A scalable map matching system for the Open Source Community. In *GI-Jahrestagung*, pages 2109–2119, 2014.

[12] P. Newson and J. Krumm. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 336–343. ACM, 2009.

[13] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[14] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 673–684. IEEE, 2002.

[15] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of the 18th international conference on World wide web*, pages 791–800. ACM, 2009.