

# Scalable Processing of Trajectory-Based Queries in Space-Partitioned Moving Objects Databases

Ralph Lange

Frank Dürr

Kurt Rothermel

Institute of Parallel and Distributed Systems

Universitätsstraße 38

70569 Stuttgart, Germany

<firstname.lastname>@ipvs.uni-stuttgart.de

## ABSTRACT

Space-partitioned Moving Objects Databases (SP-MODs) allow for the scalable, distributed management of large sets of mobile objects' trajectories by partitioning the trajectory data to a network of database servers.

Processing a spatio-temporal query  $q$  therefore requires efficiently routing  $q$  to the servers storing the affected trajectory segments. With a coordinate-based query – like a spatio-temporal range query – the relevant servers are directly determined by the queried range. However, with trajectory-based queries – like retrieving the distance covered by a certain object during a given time interval – the relevant servers depend on actual movement of the queried object. Therefore, efficient routing mechanisms for trajectory-based queries are an important challenge in SP-MODs.

In this paper, we present the Distributed Trajectory Index (DTI) that allows for such efficient query routing by creating an overlay network for each trajectory. We further present an enhanced index called DTI+S. It accelerates the processing of queries on aggregates of dynamic attributes, like the maximum speed during a time interval, by augmenting DTI with summaries of trajectory segments. Our simulations with a network of 1000 database servers show that DTI+S can reduce the overall processing time by more than 98%.

## Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Spatial databases and GIS*

## General Terms

Algorithms, design, experimentation, performance

## Keywords

Spatio-temporal indexing, moving objects database, MOD, trajectory-based query, distributed query processing

## 1. INTRODUCTION

The availability of global positioning systems like GPS together with cheap positioning devices has led to a multitude of location-based services (LBS). Advanced LBS not only consider the current positions of mobile objects (MOs) but also past positions, i.e. the *trajectories* of the MOs.

*Moving Objects Databases* (MODs) have been proposed to store and manage trajectories of MOs. In the last decade, numerous structures to efficiently access past trajectory data have been investigated [1, 3, 9–12]. However, most of them have been designed for centralized MODs, where query processing takes place on a single database server. While centralized MODs have their merits, they do not scale well with the number of MOs and the size of the service area. Distributed MODs in contrast allow the load to be distributed over multiple servers. They are suitable for a multitude of applications like toll collection, pay-as-you-drive car insurance, and fleet management.

Both, the type of supported queries and the way how trajectory data is partitioned to the servers, have a strong impact on the MOD organization.

We can distinguish between two classes of spatio-temporal queries, *coordinate-based* and *trajectory-based* queries [1, 12]. Queries of the former class refer to all trajectories satisfying a certain spatial relationship to a specified region or point. For instance, a spatio-temporal range query, which returns all MOs residing in a given region during a given time interval, belongs to that class. In contrast, queries of the latter class refer to the trajectory of a single MO and return information concerning a specified segment of that trajectory. For example, such a query may refer to the trajectory segment of a MO  $o$ , for a time interval  $[t_i, t_j]$ , delivering the segment itself or just the segment's length. Both query classes are highly relevant to location-based applications and hence should be efficiently supported by MODs.

For distributing trajectory data two kinds of partitioning are conceivable, *spatial partitioning* and *object-based partitioning*. With spatial partitioning a server stores all trajectory segments that overlap with the geographic service region associated with this server. Therefore, a MO's trajectory data is typically distributed over multiple servers. In contrast, with object-based partitioning a MO's trajectory is entirely stored on a single server, where the server responsible for a MO is determined by a well-known mapping.

A critical issue with query processing in a distributed MOD is routing a given spatio-temporal query to the servers that store the queried data. With respect to query routing, the two kinds of partitioning have different characteristics:

© ACM, 2008. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 16th ACM SIGSPATIAL Int'l Conference on Advances in Geographic Information Systems (ACM GIS '08), pp. 270–279. Irvine, CA, USA. November 2008. <http://doi.acm.org/10.1145/1463434.1463474>

1. *Object-based partitioning*: Object-based partitioning is well suited for processing a trajectory-based query since the queried trajectory is stored on a single server, directly given by the partitioning scheme and the identifier of the queried MO. On the other hand, a coordinate-based query has to be distributed to a large number of servers in general, since every server might store relevant data on the queried region and time interval.
2. *Spatial partitioning*: For a coordinate-based query, like a range query, the set of servers that store relevant data is given directly by the queried region and the mapping from space to servers. If the queried region is small compared to the servers’ service regions, then the query has to be routed to few servers only. An algorithm for efficient distributed processing of range queries has been proposed lately in [15]. On the other hand, for a trajectory-based query, the set of servers that store the queried data not only depends on the query parameters and the spatial partitioning but also on the actual route of the queried trajectory.

At first glance the characteristics of spatial partitioning seem to be complementary to the ones of object-based partitioning. Yet, there is a crucial difference in favor of spatial partitioning: With object-based partitioning potentially every server can store relevant data for a given coordinate-based query, even if the queried range is small. However, this does not apply to spatial partitioning and trajectory-based queries: For a given trajectory and a short time interval the relevant data is stored by few neighboring servers only, due to the functional dependency between space and time induced by the movement of the respective MO.

Another important advantageous characteristic of spatial partitioning in MODs with larger service areas is that current position information can be stored close to the MOs by distributing the servers according to their service regions. This may reduce the overhead for position updates substantially [6, 13]. Therefore, spatial partitioning and distribution also is widely used in scalable location management systems [6, 13, 17].

Partitioning	Object-based	Spatial
Trajectory-based queries	+	?
Coordinate-based queries	-	+
Update-aware distribution	-	+

**Table 1: Characteristics of partitioning schemes.**

Table 1 briefly summarizes the characteristics of both kinds of partitioning. The question mark in Table 1 shall indicate that spatial partitioning does neither inherently prevent efficient processing of trajectory-based queries nor that it immediately allows for an efficient processing scheme.

Therefore, in the following, we show how to efficiently support trajectory-based queries in a *Space-Partitioned MOD* (SP-MOD) using suitable distributed index structures. Our results will allow for scalable SP-MODs that provide their service on a larger scale (e.g., national or international) and support both query classes efficiently.

The main contribution of this paper is the *Distributed Trajectory Index* (DTI), enabling efficient routing of trajectory-based queries in a SP-MOD. A DTI realizes an overlay network of servers storing segments of a certain trajectory. Our evaluations show that DTI reduces the time for routing

trajectory-based queries to the relevant servers by up to 73% compared to routing without DTI.

We further propose an enhanced index called *DTI+S* optimizing aggregation queries on trajectories like “What was the maximum speed of object  $o$  during time interval  $[t_1, t_2]$ ?”. DTI+S optimizes DTI for such queries by augmenting the index with summaries of trajectory segments. Our evaluation shows that DTI+S can reduce the time for processing aggregation queries by more than 98% compared to processing without DTI+S. DTI and DTI+S achieve these savings with negligible storage consumptions compared to the trajectory data being stored. In our evaluations DTI+S accounts for less than 4.2% of the overall storage consumption.

The remainder of the paper is structured as follows: In Section 2 we present our system model. In Section 3 we discuss the general procedure of processing trajectory-based queries in a SP-MOD. In Section 4 we present DTI and the algorithms for its construction and for query routing. In Section 5 we present DTI+S and the corresponding query processing algorithm. In Section 6 we discuss the robustness of DTI+S regarding repartitioning of service regions. We show the effectiveness of DTI and DTI+S in extensive simulations in Section 7 and discuss related work in Section 8. The paper is concluded in Section 9 with a summary.

## 2. SYSTEM MODEL

Our system model consists of a SP-MOD which stores the trajectories of a set of MOs with identifiers  $o_1$  to  $o_m$ , where  $m$  is in the magnitude of thousands or millions in general.

A trajectory is a spatio-temporal polyline – i.e. a continuous, piecewise linear function from time to the X-Y plane – represented by a sequence of timestamped positions  $p_0, p_1, p_2, \dots$ , where each  $p_i$  has three attributes  $x$ ,  $y$ , and  $t$ . We refer to any connected clipping of a trajectory as *trajectory segment*. On a known trajectory such a segment simply can be specified by a time interval  $[t_i, t_j]$ .

The SP-MOD is responsible for tracking MOs in a certain geographic area, called *service area*. This area is partitioned into a set of disjoint *service regions*. Each service region is managed by an individual server  $s_i$ . Therefore, the SP-MOD is composed of a collection of servers  $s_1$  to  $s_n$ , each of which is responsible for a single service region. For any MO a server stores only positions that are located in its service region. Consequently, a MO’s trajectory generally is maintained by multiple servers, each storing the segment intersecting its service region. Also, a server only stores trajectory segments of those MOs that have visited its service region at least once. We do not make any assumptions regarding the storage and indexing of the trajectory segments within a server. Any index structure for past trajectory data like TB-tree [12] can be used.

We call the server in whose service region MO  $o$  is currently located the *current server* of  $o$ .  $S(o)$  denotes the servers storing segments of  $o$ ’s trajectory.  $S(o, t)$  denotes the server storing the segment comprising timestamp  $t$ . Two servers are *neighbors* if their service regions adjoin. Figure 1 illustrates a SP-MOD with seven servers  $s_1$  to  $s_7$ .

We assume that the servers compose a geographic overlay network with respect to their service regions. Given a message and a target point  $(x, y)$  they are able to route the message geographically from any server to the server whose service region contains  $(x, y)$ . We do not make any further assumptions about how geographic routing is implemented.

The current positions of the MOs can be determined by any kind of position sensors located on the MOs – such as GPS receivers – or within the infrastructure surrounding the MOs. We assume that there exists an appropriate position update protocol [5] – possibly employing dead reckoning techniques – which delivers a MO’s current position from the position sensor to the corresponding server. If a MO moves from one service region to another, the update protocol performs a *handover* which supplies the address of the MO’s previous current server to the new one.

Any entity that is connected to the network of servers can act as a *client* of the SP-MOD. A client can issue a query at an arbitrary server. The servers process the query in a distributed fashion and transmit the result to the client.

The SP-MOD is capable of processing coordinate- and trajectory-based queries. However, in this paper, we address the latter class of queries only. An algorithm for processing range queries in a SP-MOD is given in [15]. As stated above, trajectory-based queries refer to the trajectory segment of a single MO. In particular, a query  $q_{\text{type}}(o, t_s, t_e)$  refers to the segment specified by time interval  $[q.t_s, q.t_e]$  of MO  $q.o$ . The type of the query defines what information concerning this *queried segment* is returned to the client. Due to space limitations we confine ourselves to four important types:

1. *Segment query*: the specified segment is returned as a list of positions.
2. *Length query*: the length of the specified segment is returned.
3. *Max-speed query*: the maximum speed of  $q.o$  in the specified segment is returned.
4. *Region-relation query*: This type of query takes an additional parameter  $R$  defining a geographic region. The query returns the set of time intervals within the queried interval  $[q.t_s, q.t_e]$  during which  $q.o$  was located inside  $q.R$ .

In the following,  $S(q)$  denotes the servers storing the queried segment of  $q$ . Note that  $S(q) \subseteq S(q.o)$ .

### 3. BASIC SCHEME

In this section we introduce a basic scheme for processing trajectory-based queries in SP-MODs. In the subsequent two sections we improve this scheme by DTI and DTI+S.

A client may send a query  $q$  to any server. Unfortunately, due to the spatial partitioning there exists no immediate mapping from the parameters  $q.o$ ,  $q.t_s$  and  $q.t_e$  to the servers  $S(q)$ . Therefore, we need a mechanism for routing  $q$  to  $S(q)$ .

With our basic scheme, query routing comprises three phases: (1.) Routing  $q$  to any server  $s_i \in S(q.o)$ , (2.) routing  $q$  to  $S(q.o, q.t_s)$  or  $S(q.o, q.t_e)$ , i.e. to the start or end of the queried segment, and (3.) routing  $q$  to the remaining servers of  $S(q)$ , i.e. traversing the queried segment to collect the queried information.

#### 3.1 First Phase

A server  $s_i \in S(q.o)$  can be found by potentially searching all servers of the SP-MOD, which obviously does not scale well. Instead we apply a *home server* scheme, where each MO  $o$  has a home server that knows (at least) one server of  $S(o)$ . Our approach is inspired by distributed hash tables, which are used to (indirectly) assign home server roles to the various servers of the SP-MOD. For that purpose, all servers share a fixed hash function  $f_h$  mapping the MOs’ identifiers to points in the service area:  $f_h : \{o_1, \dots, o_m\} \rightarrow \mathbb{R}^2$ . The

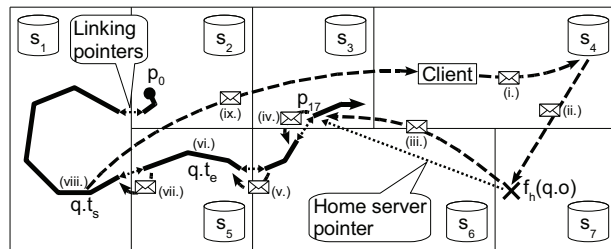


Figure 1: Query processing using the basic scheme.

home server of MO  $o$  is defined to be the server whose service region contains the geographic point  $f_h(o)$ . This server maintains a *home server pointer*  $p_h$  to some timestamped position  $p_i$  of  $o$ ’s trajectory. The pointer  $p_h$  simply is a *copy* of its target  $p_i$ , i.e. it is independent of the server storing  $p_i$ . This independence has the advantage that home server pointers are stable to reconfigurations, i.e. need not be updated if their target positions are relocated to other servers.

How are the targets of home server pointers selected? A simple mechanism is to select the MO’s initial position  $p_0$ . Of course we can think of much more elaborate schemes, which adapt the pointer to frequently queried time intervals or which maintain more than one pointer. Since the DTI scheme is independent of any particular solution to that problem we do not go into further detail here.

The server that first receives a query  $q$  from the client geographically routes  $q$  to  $f_h(q.o)$ , i.e. to the home server of  $q.o$ . The home server then geographically routes the query to  $(p_h.x, p_h.y)$ . This completes the first phase. If any server  $s_i \in S(q.o)$  receives  $q$  on its way to  $(p_h.x, p_h.y)$  the first phase is completed even earlier.

#### 3.2 Second Phase

In the second phase  $q$  is routed to the server storing the beginning or end of the queried segment, i.e. to  $S(q.o, q.t_s)$  or to  $S(q.o, q.t_e)$ . Without loss of generality we only consider routing to  $S(q.o, q.t_e)$  in the following.<sup>1</sup> Each server that receives  $q$  transmits it to the neighbor server storing the previous or rather subsequent segment that is closer to  $q.t_e$  until  $S(q.o, q.t_e)$  is reached. We refer to this kind of linear routing along a trajectory as *trajectory-based routing*.

For this purpose the servers maintain two *linking pointers* for each segment they store: The *backward linking pointer* points to the last position of the previous segment and the *forward linking pointer* points to first position of the subsequent segment. Like a home server pointer a linking pointer is a copy of the timestamped position it points to. This enables each server to locally decide on the next routing step.

The linking pointers are created as follows: For each MO the SP-MOD maintains a data structure called *shadow object* (SO). The SO for a MO  $o$  is transferred from the current server of  $o$  to the next one as part of the handover procedure. Besides other information the SO contains the latest known position of  $o$ . When the update protocol signals a handover (cf. Section 2) the new current server informs the previous current server on the new position of  $o$ . The previous server creates the forward linking pointer. Then it transmits the

<sup>1</sup>The improved scheme DTI+S requires that  $q$  is routed to  $S(q.o, q.t_e)$  as it processes  $q$  backwards in time from  $q.t_e$  to  $q.t_s$ , cf. Section 5.

SO to the new current server. This server creates the backward linking pointer and then updates the SO accordingly.

### 3.3 Third Phase

In the third phase  $q$  is monotonously processed from  $q.t_e$  to  $q.t_s$  using trajectory-based routing. The server  $S(q.o, q.t_e)$  processes  $q$  backwards in time until the end of the previous segment given by the backward linking pointer. Then it transmits  $q$  and the partial result  $r$  to the corresponding neighbor server. This server further processes  $q$  on its segment until the end of its previous segment and merges its local result with  $r$ . This procedure is repeated until  $S(q.o, q.t_s)$  receives  $q$ . This server processes  $q$  from the end of its segment to  $q.t_s$  and merges the local result with  $r$ . Then it sends the final result  $r$  to the client.

The type of  $r$  and its merging with local results depend on the query type. For a length query  $r$  is the length of the segment processed so far. Local results simply are added to  $r$ . Similar applies to the other types of queries. Note that the result of a segment query is not an aggregated value but simply a copy of the queried segment. Therefore the servers  $S(q)$  alternatively can send their local results to the client directly instead of concatenating them in  $r$ .

Figure 1 illustrates the three phases: (i.)  $s_4$  receives the query  $q$  from the client. (ii.)  $s_4$  geographically routes  $q$  to  $f_h(q.o)$ , i.e. to  $s_7$ . (iii.)  $s_7$  routes  $q$  to  $(p_{17}.x, p_{17}.y)$  according to its home server pointer  $p_h = p_{17}$ . (iv.) The respective server  $s_3$  sends  $q$  to its neighbor server  $s_6$  as the queried time interval  $[q.t_s, q.t_e]$  is before  $s_3$ 's segment. (v.)  $s_6$  sends the query to  $s_5$  for the same reason. (vi.)  $s_5$  starts processing  $q$  as its segment comprises  $q.t_e$ . (vii.) Then it sends  $q$  and the partial result  $r$  to its neighbor  $s_1$ . (viii.) This server completes the processing since its segment comprises  $q.t_s$ . (ix.) Finally  $s_1$  sends  $r$  to the client.

## 4. DISTRIBUTED TRAJECTORY INDEX

Trajectory-based routing in the second phase can take a lot of geographic routing hops and thus time. It depends on the trajectory's route, the service regions and the *temporal routing distance*, i.e. the time span between the segment of the first server of the second phase and  $q.t_e$ . For example, for uniform movement and uniform service regions the number of hops linearly depends on the temporal routing distance.

The goal of the DTI scheme is to increase the efficiency of routing by creating a distributed index called DTI over the servers  $S(o)$  of each trajectory. DTI stores additional pointers along the trajectory. These *DTI pointers* allow for direct routes from one server's segment to temporal distant segments stored by other servers. Thus, a DTI realizes an overlay network of servers in  $S(o)$ .

The DTI scheme employs the idea of maintaining multiple pointers spanning different distances from skip lists [14] and applies it to distributed indexing of trajectory data. More precisely, a DTI is created according to a perfect bidirectional skip list, illustrated in Figure 2a. The nodes of such a list are sequentially numbered starting at zero. The pointers have different *levels*. The pointers at level 0 compose a doubly-linked list over all nodes. The pointers at level 1 compose a doubly-linked list over all nodes with even sequence numbers. The pointers at level  $i$  compose a doubly-linked list over all nodes with sequence numbers divisible by  $2^i$ . The *height* of a node is the maximum level of the pointers it maintains. Thus, the height  $l$  of a node with sequence

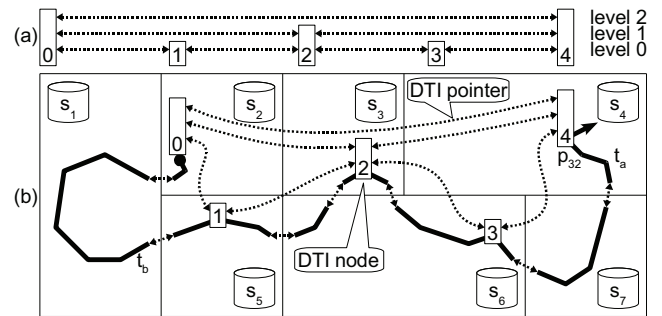


Figure 2: Skip list and DTI.

number  $k > 0$  is  $l = \max \{i : 2^i | k\}$ . The height of node 0 is always equal to the maximum height of all other nodes.

Figure 2b illustrates how the skip list principle is applied to a DTI for a given trajectory. The DTI scheme selects individual positions to act as *anchors* of list nodes. The DTI implements such a logical node by a data record named *DTI node*. A DTI node is maintained by the server that stores the node's anchor – e.g.,  $s_4$  stores node 4 with anchor  $p_{32}$ .

A DTI node contains its sequence number, its anchor, and the pointers to other DTI nodes. Such a *DTI pointer* simply is a copy of the anchor of the node it refers to. Thus, the DTI pointers are realized like linking pointers and home server pointers. Therefore, trajectory-based routing in the second phase of the basic scheme can be adapted easily to the more efficient *DTI-based routing*, explained below.

A DTI is extended incrementally during runtime: Once the time span since the creation of the last DTI node exceeds a certain threshold  $T_R$  (e.g., one hour, cf. Section 7.2), the current server of the respective MO  $o$  creates a new node when receiving the next position update from  $o$ . Therefore, a server can store no DTI nodes at all as well as several DTI nodes regarding a certain segment. The creation of DTI nodes is explained in detail in Section 4.2.

The distributed storage of the DTI nodes according to their anchors is useful since it provides direct network locality between index and indexed data. Furthermore, a DTI is robust regarding repartitioning of service regions due to the indirect addressing based on timestamped positions instead of network addresses, c.f. Section 6. Hence, a DTI realizes a *lightweight* overlay network on top of geographic routing, which again can be realized as an overlay network.

The overall storage consumption of a DTI linearly depends on its number of DTI nodes since the expected number of DTI pointers per node is *independent* of the number of DTI nodes. This can be seen from the following series: On average, half of the nodes maintain two pointers, one fourth maintain four pointers, one eighth maintain six pointers, and so on. Thus, the expected number of pointers at an arbitrary DTI node is equal to

$$\frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 6 + \dots + \frac{1}{2^i} \cdot 2i + \dots = \lim_{n \rightarrow \infty} 2 \sum_{i=1}^n \frac{i}{2^i} = 4.$$

Accordingly, the expected height of an arbitrary DTI node is  $l = 1$ .

### 4.1 DTI-based Routing

The fundamental principle of DTI-based routing is *greedy temporal routing*: Given a query  $q$ , each server geographi-



cally routes  $q$  to the target of the pointer  $p$  that minimizes  $|p.t - q.t_e|$ . Algorithm 1 gives a formal description of the algorithm executed at each server. The algorithm not only transmits  $q$  but also the pointer  $p$  currently being used as geographic routing along a DTI pointer may encounter servers that do not store any information of the queried trajectory.

---

**Algorithm 1** DTI-based routing

---

```

1: receive query  $q$  and pointer  $p$ 
2: if stores segment that comprises  $q.t_e$  then
3:   Enter third phase of basic scheme . . .
4: else
5:   if maintains any pointer on  $q.o$ 's trajectory then
6:      $p' \leftarrow$  pointer that is temporally closest to  $q.t_e$ 
7:     if  $|p'.t - q.t_e| < |p.t - q.t_e|$  then
8:        $p \leftarrow p'$ 
9:     end if
10:  end if
11:  geographically route  $q$  and  $p$  towards  $(p.x, p.y)$ 
12: end if

```

---

If a server receives  $q$  and  $p$  it first checks whether it stores the segment that comprises  $q.t_e$  (line 2). If not, it selects the pointer  $p'$  that is temporally closest to  $q.t_e$  from *all* pointers known to it – in particular DTI pointers, but also linking pointers and possibly even the MO's home server pointer (line 6). If  $p'$  is even closer to  $q.t_e$  than  $p$ , it replaces  $p$  accordingly (lines 7 – 8). Finally it geographically routes  $q$  and  $p$  towards  $p$ 's target. For example, if server  $s_3$  in Figure 2b receives a query with  $q.t_e = t_a$ , then DTI-based routing uses the forward DTI pointer from DTI node 2 to node 4 and thus saves two geographic routing hops compared to trajectory-based routing.

The routing algorithm above generally achieves a good routing performance due to the following advantageous properties of a perfect bidirectional skip list:

1. The bidirectionality allows for efficient routing forward and backward in time.
2. The levels guarantee that a server always maintains more DTI pointers to temporally close segments than to distant ones. Hence, the smaller the temporal distance to  $q.t_e$ , the finer is the support by pointers.
3. The number of DTI pointers for routing from one DTI node to another node logarithmically depends on the temporal distance of the two nodes.

The importance of the latter two properties for efficient routing in overlay networks is well known from the seminal work of Kleinberg [4]. The performance of DTI-based routing depends on the actual route of the trajectory, the performance of the underlying geographic routing algorithm, and the temporal density of the DTI nodes given by  $1/T_R$ . For a given  $T_R$ , uniform movement, and a fully meshed geographic overlay network, the number of geographic routing hops with DTI-based routing logarithmically depends on the temporal routing distance. It polylogarithmically depends on the temporal routing distance in case of a geographic overlay network with long-range links according to Kleinberg [4].

The choice of  $T_R$  is a trade-off between the overhead for storing and accessing the DTI pointers within a server and the performance improvement for query routing. However, our evaluation results in Section 7.2 show, that suitable values for  $T_R$  range within an order of magnitude.

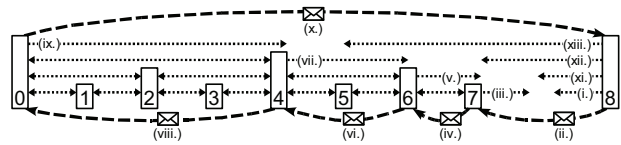


Figure 3: Creation of DTI node 8.

The routing performance particularly improves with *implicit shortcuts*. That is, a server stores more than one segment of the same trajectory and thus enables to jump locally to a temporally distant segment. At the same time implicit shortcuts show that the above algorithm only approximates the optimal routing path. Each server chooses the temporally closest pointer independent of whether the server at the pointer's target provides implicit shortcuts or not.

A second reason for non-optimal routing paths is illustrated in Figure 2b. If  $s_2$  receives a query with  $q.t_e = t_b$  then using the DTI pointer from DTI node 0 to 1 results in an additional geographic routing hop compared to routing along the forward linking pointer from  $s_2$ 's segment to  $s_1$ 's segment. The reason is that the temporal distance to  $q.t_e$ , used for choosing the next pointer, only approximates the actual routing distance.

## 4.2 Creation of DTI Nodes

Once the time span since the creation of the last DTI node exceeds a certain threshold  $T_R$ , the current server of the respective MO  $o$  creates a new node when receiving the next position update from  $o$ . For this purpose the DTI scheme stores the sequence number and anchor of the last DTI node in the SO. If the current server creates a new DTI node, it determines the sequence number  $k$  of the new node and the backward DTI pointer to node  $k-1 = k-2^0$  at level 0. Then it creates a *DTI-pointer message* which contains  $k$  and the new DTI node's anchor –  $o$ 's current position given in the update message. It geographically routes the message to the server maintaining node  $k-1$ . This server creates the opposite forward DTI pointer at level 0. If the new node's height  $l$  equals zero, then the server immediately acknowledges the DTI-pointer message. If  $l \geq 1$ , it adds the backward DTI pointer to node  $k-2$  to the message and routes the message to this pointer's target. The server that maintains DTI node  $k-2$  creates the forward DTI pointer at level 1 to node  $k$ , adds its backward DTI pointer to  $k-4 = k-2^2$  to the message, and then routes the message to node  $k-4$ . This procedure is repeated on ascending levels until node  $k-2^l$  is reached. The server of node  $k-2^l$  creates the forward DTI pointer on level  $l$  to the new node  $k$  and then routes the message back to the current server of the MO. With it the current server creates the new node's backward DTI pointers on the levels 1 to  $l$ . Thus, the creation of a new DTI node with height  $l$  involves routing the DTI-pointer message geographically to at most  $l+1$  other servers and creating  $2l+2$  DTI pointers, where  $l=1$  on average, see above.

Figure 3 illustrates this procedure for creating DTI node 8. The Roman numbers denote the chronological order of the creations of the DTI pointers and the intermediate transmissions of the DTI-pointer message.

## 4.3 Home Server Pointer and DTI

DTI-based routing works if the home server pointer refers

to any position on the MO’s trajectory. However, in order to reduce the average routing time it is advantageous if the home server pointer points to the anchor of a DTI node with a large number of DTI pointers. The DTI scheme guarantees that the home server pointer always points to the highest DTI node  $k > 0$  by updating the home server pointer each time it creates a DTI node with  $k = 2^l$ .

## 5. ENHANCING DTI WITH SUMMARIES

In the previous section we introduced the DTI scheme to reduce the routing overhead for the second phase of query processing. However, it does not affect the third phase, where trajectory-based routing may still cause a substantial routing overhead. Obviously, this overhead depends on the time span  $q.t_e - q.t_s$ . In the following, we extend the DTI scheme to increase the routing and thus query processing performance for the third phase.

The basic idea of the extended scheme, called DTI+S, is to attach *summaries* to the DTI pointers. A summary attached to a DTI pointer records aggregated information concerning the segment between the anchor of the respective DTI node and the pointer’s target. For example, a summary may store the length of this segment and the maximum speed recorded for this segment. Summaries may substantially speed up processing in the third phase. In many cases, queries can be forwarded along DTI pointers using the aggregates stored in the attached summaries. Without those summaries this information would have to be aggregated by additionally visiting all other servers of  $S(q)$ .

This approach can be applied to *any* type of query on an aggregate of a dynamic attribute that can be computed by partial aggregation [8], i.e. by means of smaller aggregates. In particular it can be applied to length, max-speed, and region-relation queries. For these queries, each summary stores the length of the segment it refers to, the maximum speed within the segment, and the minimum bounding rectangle (MBR) of the segment, respectively. Clearly, summaries cannot be used for optimizing segment queries as the result of such a query is not an aggregated value.

In principle, summaries can be attached to forward and backward DTI pointers. Whether summaries are maintained for a single or both directions is a trade-off between storage consumption and routing performance. In the following we assume that summaries are only stored with the backward pointers. Therefore query processing in the third phase always starts at  $q.t_e$ . However, the scheme can be easily extended to provide summaries for both directions.

The summaries increase the average storage consumption per DTI node by a constant amount of data. Assuming that storing a timestamped position requires  $3 \times 8 = 24$  byte and that an integer takes 4 byte, a DTI node without summaries but with anchor, sequence number, and DTI pointers – four on average, cf. Section 4 – consumes  $24 + 4 + 4 \times 24 \approx 125$  byte. For the three kinds of aggregation queries we consider in this paper a summary contains six floating-point values, i.e. it requires  $6 \times 8 = 48$  byte. Thus, a DTI node with summaries consumes  $125 + 2 \times 48 \approx 225$  byte on average.

### 5.1 Construction of DTI+S

The DTI+S scheme creates the summaries together with the backward pointers. For this purpose each SO additionally contains a summary of the segment between the latest DTI node and the latest known position. The current server of

---

### Algorithm 2 DTI+S-based query routing and processing

---

```

1: receive query  $q$ , pointer  $p$  and partial result  $r$ 
2: while  $p.t > q.t_s$  do
3:   if has usable summary to cut short from  $p.t$  then
4:     determine summary that enables longest shortcut
5:     merge  $r$  with aggregate given in the summary
6:      $p \leftarrow$  backward DTI pointer of the summary
7:   else if has segment that spans  $p.t$  then
8:      $p' \leftarrow$  linking pointer to previous segment
9:     if has DTI node between  $p'.t$  and  $p.t$  then
10:       $p' \leftarrow$  anchor of latest such DTI node
11:     end if
12:     if  $q.t_s > p'.t$  then
13:        $p' \leftarrow$  interpolate timestamped position at  $q.t_s$ 
14:     end if
15:     process  $q$  on local segment between  $p'.t$  and  $p.t$ 
16:     merge  $r$  with result between  $p'.t$  and  $p.t$ 
17:      $p \leftarrow p'$ 
18:   else
19:     geographically route  $q$ ,  $p$ , and  $r$  towards  $(p.x, p.y)$ 
20:     return
21:   end if
22: end while
23: send  $r$  to client

```

---

the MO updates this summary with each position update. For example, in case of the segment length it increments the length given in the summary by the distance between the new position given in the position update message and the previous position.

If the current server creates a new DTI node with sequence number  $k$  the SO’s summary yields the summary for the segment between the DTI node  $k - 1$  and the new node  $k$ , i.e. it belongs to the new backward DTI pointer at level 0. The summaries belonging to the backward pointers on higher levels are created by concatenating the SO’s summary with summaries for segments between previous DTI nodes.

For this purpose the DTI-pointer message for a new node  $k$  with height  $l$  is extended as follows: The servers of the nodes  $k - 2^0, k - 2^1, \dots, k - 2^{l-1}$  not only add the backward DTI pointer to the next node in the list to the message but also add the corresponding summary. When the server of the new node  $k$  finally receives the message it computes the summaries for the backward pointers at the levels 1 to  $l$ : For the summary attached to the pointer at level 1 it concatenates the SO’s summary and the summary on the segment between node  $k - 2^0$  and node  $k - 2^1$ . For the summary at level 2 it concatenates the just created summary and the summary on the segment between the nodes  $k - 2^1$  and  $k - 2^2$ , and so on.

Two consecutive summaries are concatenated by aggregating each pair of aggregates belonging together. For example, the length of the segments simply are added.

### 5.2 DTI+S-based Routing and Processing

Routing with DTI+S in the third phase is also based on *greedy temporal routing*. However, a given query  $q$  only may be routed along a backward DTI pointer if the pointer’s summary is *usable* for processing  $q$ . That is, the summary provides the needed information for processing  $q$  on the summary’s time interval.

Whether a summary can be used depends on the query

type. Therefore, we first explain the generic algorithm for DTI+S-based routing and processing. Then, we render the algorithm more precisely by discussing usable summaries for the types of aggregation queries considered in this paper.

Algorithm 2 shows the generic algorithm executed at each server. It monotonously processes  $q$  from  $q.t_e$  to  $q.t_s$  backwards in time. The current progress is stored by the timestamp  $p.t$  of the pointer  $p$ . Initially  $p$  is equal to the position at  $q.t_e$ .

A server  $s_i$  that receives  $q$ ,  $p$  and  $r$  repeatedly tries to process  $q$  from  $p.t$  on. In each repetition it first tries to process  $q$  by means of a usable summary (line 3 – 6) and otherwise by the segment that comprises  $p.t$  (lines 7 – 17). If it neither stores a usable summary nor the segment that comprises  $p.t$ , then it routes  $q$  towards the server of the remaining part of the queried time interval (lines 18 – 19). If the server completed query processing ( $p.t \leq q.t_s$  in line 2), it sends the aggregated result to the client.

If the server maintains any usable summaries, it greedily processes  $q$  by means of the one reaching furthest in the past. If it has no usable summary but stores the segment spanning  $p.t$ , then it processes  $q$  by means of this segment. At every visited DTI node,  $s_i$  tries again to find a usable summary by repeating the while loop instead of simply traversing the segment linearly.

Whether a summary can be used depends on the query type. However, there exists a necessary condition that applies to all query types: A usable summary’s time interval  $[t_i, t_j]$  must contain  $p.t$ , i.e.  $t_i < p.t \leq t_j$ . For other summaries the above algorithm cannot advance  $p.t$  towards  $q.t_s$ .

Next, we define the sufficient conditions for usable summaries for the aggregation queries we consider in this paper:

- *Length queries*: A summary is usable if and only if its time interval  $[t_i, t_j]$  holds for  $q.t_s \leq t_i$  and  $p.t = t_j$ . Thus, a usable summary always belongs to the DTI node with anchor  $p.t$ .
- *Max-speed queries*: A summary is usable iff it fulfills one of the following two conditions: (1.) The summary’s time interval  $[t_i, t_j]$  holds for  $q.t_s \leq t_i \leq p.t$  and  $p.t \leq t_j \leq q.t_e$ . (2.) The summary holds for the necessary condition and the maximum speed given in the summary is less or equal  $r$ . For this reason – compared to length queries – the algorithm sometimes can even use a summary on a time interval  $[t_i, t_j]$  with  $t_i < q.t_s$  or  $t_j > q.t_e$ .
- *Region-relation queries*: A summary is usable iff it fulfills the necessary condition above and its MBR shows that the corresponding segment is located completely inside or completely outside  $q.R$ . That is,  $q.R$  either completely contains the MBR or does not intersect the MBR. If the MBR only partially overlaps  $q.R$ , the summary is unusable. However, in this case the query possibly may be processed by a sequence of summaries on sub-segments of this summary’s segment, since the MBRs of two consecutive segments generally cover much less space than the MBR of both segments.

## 6. SERVICE REGION REPARTITIONING

To achieve a scalable SP-MOD service, it is essential that service regions can be split and merged locally without global reconfiguration. That is, splitting and merging must only affect a small number of servers. The basic scheme as well as DTI and DTI+S meet this requirement well: Anchors of

DTI nodes and all kinds of pointers base on timestamped positions rather than addresses of servers.

Splitting the service region of a server  $s_i$  with a new server  $s_j$  therefore only affects segments, pointers and DTI nodes of  $s_i$ . In detail, the following five tasks have to be performed: (1.) Migrating each SO from  $s_i$  to  $s_j$  whose MO is located in  $s_j$ ’s service region. (2.) Splitting the segments stored by  $s_i$  according to the new service regions and creating linking pointers at the resulting splits. (3.) Migrating the segments that are located in  $s_j$ ’s service region to  $s_j$  together with their linking pointers. (4.) Moving each home server pointer of MO  $o$  from  $s_i$  to  $s_j$  where  $f_h(o)$  is located in the service region of  $s_j$ . (5.) Migrating the DTI nodes, whose anchors are located in  $s_j$ ’s service region, and their summaries to  $s_j$ .

A similar procedure applies to merging two neighboring service regions.

## 7. EVALUATION

In this section, we evaluate the effectiveness of DTI and DTI+S for query routing and processing in SP-MODs. First, we explain the simulation setup. Then, we discuss the performance of DTI-based routing by considering the processing time in the first and second phase only. Finally, we give results on the overall processing time with DTI+S.

### 7.1 Simulation Setup

We implemented an extensive discrete event simulator for SP-MODs which allows for measuring the query processing times resulting from network latencies, disk I/O, and CPU.

We conducted a series of experiments, simulating a SP-MOD with a service area of 4500 km  $\times$  2000 km ( $\approx$  Continental U.S.) over a time of  $3 \cdot 10^7$  s  $\approx$  1 year. If not stated otherwise, the SP-MOD is composed of 1000 servers with rectangular service regions with side lengths between 140 and 560 km. The network latencies for query routing are modeled using a real topology data set of the AT&T internet backbone [7]. Each server of the SP-MOD is connected to the router that is closest to the center of its service region assuming a network link with a delay of 3 ms. Then, the latencies are calculated using Dijkstra’s algorithm.

For geographic routing of queries and other messages each server maintains a shortest-path routing table containing *contacts* – network address and service region – for all neighbor servers. If not stated otherwise, the routing table further contains ten *Long-Range Contacts* (LRCs) to distant servers, randomly chosen using inverse square distribution [4]. If a server receives a message with target  $(x, y)$  then it greedily chooses the contact whose service region is closest to  $(x, y)$  and sends the message to the respective server.

Each server maintains a table for permanent storage of

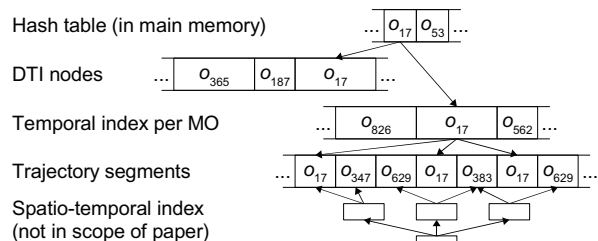


Figure 4: Storage layout.

the trajectory segments within its service region and their linking pointers as well as a table for the respective DTI nodes and summaries. The latter table is clustered by the MOs’ identifiers. Hence, all DTI nodes and summaries of a given MO can be read in a single operation. We do not make any assumptions on the clustering of the trajectory segments table. Particularly, it can be clustered to optimize processing of coordinate-based queries using any appropriate index, like TB-tree [12]. We only assume – like with TB-tree – that each page contains consecutive timestamped positions of a single MO only. For efficient trajectory-based access to the trajectory segments table, a server maintains small, sparse temporal indexes for each MO that ever entered its service region. These temporal indexes also are permanently stored, clustered by the MO’s identifiers. Figure 4 illustrates this storage layout. A hash table in main memory provides initial access to the pages that contain the DTI nodes and the temporal index of a given MO.

In our simulations we assumed a page size of 4 kB, a seek time of 10 ms and a transfer rate of 30 MB/s. With these values and the above-mentioned storage requirements for the DTI nodes and summaries, 24 byte for a timestamped position and  $8 + 4 + 8 = 20$  byte for an entry in the temporal index, the disk I/O times can be calculated.

Each server further maintains a hash table for SOs and a hash table for home server pointers, not shown in Figure 4. Due to the frequent accesses to this data and the small storage requirements (100 byte per SO and 24 byte per home server pointer) both can be cached in main memory.

The CPU times for processing within a server were measured during our experiments on an AMD Opteron Linux Server with 2.8 GHz and 4 GB RAM. Yet, the following results show that they are negligible compared to the network latencies and disk I/O times.

Regarding the processing of trajectory-based queries the mentioned storage layout is independent of the overall number of MOs. Therefore, only a small set of MOs has to be simulated. In our experiments we simulated ten MOs moving according to the mobility model proposed in [2]: Each MO starts at a random position with random direction and random speed  $v \leq v_{\max} = 10$  m/s. Every 10 s it randomly changes its direction between  $-0.1$  and  $+0.1$  rad and its speed between  $-3$  and  $+3$  m/s. In contrast to [2] the movement is not wrapped at the borders of the service area but reflected. The MOs report their positions using linear dead reckoning [5] with an accuracy of 10 m. During simulation, this results in  $1.88 \cdot 10^6$  update messages per MO.

For  $2 \cdot 10^7$  s  $\approx 8$  months the MOs only report their positions. Then, for  $10^7$  s  $\approx 4$  months, each MO additionally poses queries according to a Poisson process with rate  $\lambda = 0.01$  s $^{-1}$ . Thus, the simulation results are the averages of about  $10^6$  queries. We simulated a uniform mix of segment, length, max-speed, and region-relation queries. A MO  $o$  randomly chooses the parameters of a query  $q$  as follows:

- $q.o$ : Half of the queries refer to the client’s trajectory, i.e.  $q.o = o$ . The other half refers to any other MO.
- $q.t_e$ : The queried time interval ends 1000 to  $10^7$  s before current time  $t$ , where  $\log[t - q.t_e]$  is uniformly distributed. Thus, queries on the near past are more likely.
- $q.t_s$ : The queried time interval starts 1000 to  $10^7$  s before  $q.t_e$ , where  $\log[q.t_e - q.t_s]$  follows uniform distribution. Thus, queries on long time intervals are rare.

In case of a region-relation query,  $q.R$  is a quadratic region

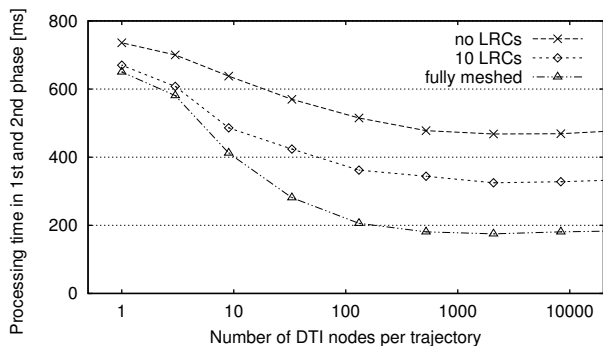


Figure 5: Routing time against DTI nodes.

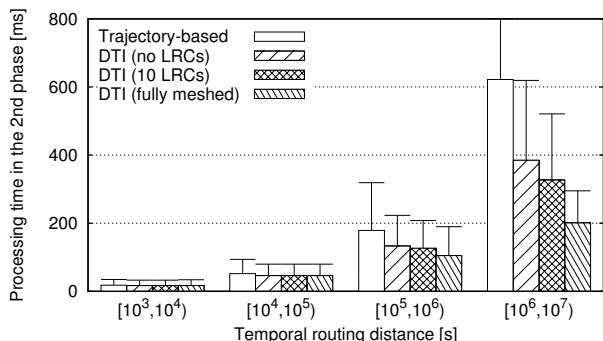


Figure 6: Routing time against routing distance.

with  $l = 10$  to  $1000$  km side length, where  $\log l$  is uniformly distributed.  $q.R$  is randomly placed in the service area.

## 7.2 DTI-based Routing

To evaluate DTI-based routing independent of summaries, we only consider the first and second phase of query processing. For readability, we refer to the processing time in these two phases as *routing time* in the following.

Figure 5 shows the average routing time using the DTI scheme depending on the number of DTI nodes per trajectory created within simulation time, i.e. depending on the temporal density of the DTI nodes determined by  $1/T_R$ . For that purpose,  $T_R$  was varied between  $900$  and  $6 \cdot 10^7$  s. Figure 5 shows three curves: one for geographic routing without LRCs, with 10 LRCs, and with LRCs to all other servers – i.e. a fully meshed geographic overlay network. In the latter case, geographic routing to any point  $(x, y)$  only requires one geographic routing hop.

With one DTI node per trajectory, DTI-based routing degenerates to trajectory-based routing. In this case, LRCs are useless except for the first phase, i.e. routing to the home server and along the home server pointer. Therefore the routing times with one DTI node are very similar for the different numbers of LRCs, between  $650$  and  $736$  ms. These times seem small compared to the fact that each trajectory is partitioned to about  $1925$  segments on average, distributed to the major part of the servers. The reason is that trajectory-based routing also uses implicit shortcuts on servers storing two or more segments of the queried trajectory, cf. Section 4.

Nevertheless, with increasing numbers of DTI nodes the



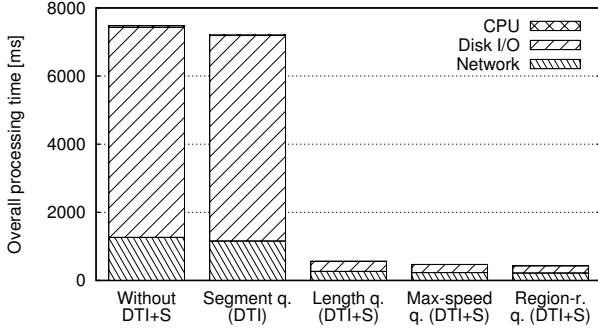


Figure 7: Itemized processing time per query type.

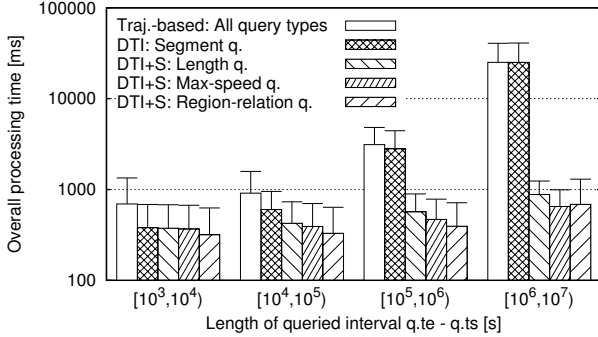


Figure 8: Processing time against queried time.

routing times significantly decrease. As expected, the more LRCs each server maintains, the larger are the savings – up to 35% (no LRCs), 51% (10 LRCs), or 73% (fully meshed). The maximum savings are reached with 1000 to 10000 DTI nodes per trajectory, i.e. with  $T_R = 30000$  to 3000s. For more than 10000 DTI nodes the routing times slightly increase due to the disk I/O for reading the additional DTI nodes. This shows, that only one DTI node per hour and trajectory is sufficient for efficient DTI-based routing in our scenario. It further shows, that the optimal routing performance is achieved also if  $T_R$  varies within an order of magnitude.

In all subsequent simulations, we used  $T_R = 1$  h, resulting in 8334 DTI nodes per trajectory within simulation time.

Figure 6 depicts the routing time in the second phase over the temporal routing distance, i.e. the time-span to cover in the second phase. As expected, DTI does not improve routing for short temporal routing distances. For temporal distances between 1000 and  $10^5$  s routing with the basic scheme as well as routing with the DTI scheme both require only about 17 ms in the second phase. Yet, for long distances –  $10^6$  to  $10^7$  s – and a fully meshed geographic overlay network the DTI scheme reduces the routing time by 67%. With 10 LRCs per server it reduces the second phase by 47% and even without LRCs the DTI scheme saves 38%.

### 7.3 DTI+S-based Routing and Processing

Next, we evaluate the DTI+S scheme by measuring the *overall processing time*, i.e. the sum of processing times in all three phases. Note that this time does not include the transmission of the result to the query issuer but only the

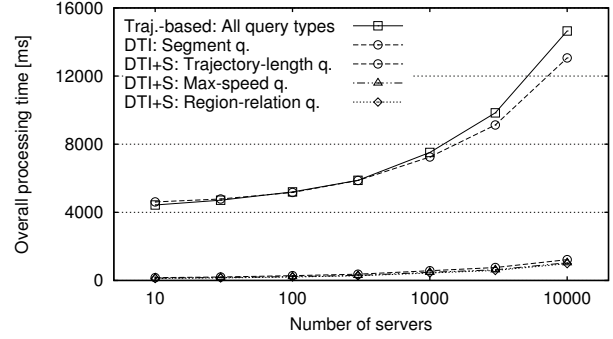


Figure 9: Processing time against number of servers.

processing within the SP-MOD.

Figure 7 shows the overall processing times with and without DTI+S for the different query types itemized to network latencies, disk I/O times, and CPU times. The figure shows only one column for query processing without DTI+S since thereby the network latencies and disk I/O times are identical for all query types and since the CPU times always are negligible compared to the other two values. In detail, without DTI+S, the average CPU time for query processing varies between 16 ms for a segment query and 68 ms for a max-speed query. With DTI+S the CPU time is 14 ms for a segment query and about 1 ms for the other query types.

Without DTI+S and for segment queries the disk I/O times of about 6100 ms clearly exceed the network latencies of about 1200 ms. With the use of summaries – i.e. for length, max-speed or region-relation queries – they both range between 200 and 300 ms. Hence, the summaries reduce the network latencies by more than 75% and the disk I/O times by even more than 95%.

Figure 8 shows the overall processing time for different lengths of the queried time intervals  $q.t_e - q.t_s$  and for different query types. Without DTI+S it linearly increases with the length of the queried time interval since implicit shortcuts cannot be utilized in the third phase. For a query on weeks or months –  $10^6$  to  $10^7$  s – processing without DTI+S requires more than 25000 ms on average. With DTI, also segment queries show a linear increase of the processing time since segment queries are not improved by summaries.

For the other three types of queries the DTI+S scheme saves large amounts of time:

- *Length queries:* Processing a length query with  $10^6 \text{ s} \leq q.t_e - q.t_s < 10^7 \text{ s}$  requires 375 ms on average.
- *Max-speed queries:* With  $10^6 \text{ s} \leq q.t_e - q.t_s < 10^7 \text{ s}$  they only require 367 ms on average as with these queries more summaries are usable than with length queries. Hence, the DTI+S scheme can reduce the overall processing time by more than 98%.
- *Region-relation queries:* For small queried time intervals DTI+S-based processing requires slightly less time than for the other types of queries since with a region-relation query the second phase can end prematurely if a server  $s_i \in S(q.o) \setminus S(q)$  with a usable summary is encountered. With longer time intervals, processing of region-relation queries requires more time than processing of length or max-speed queries since the MBRs of summaries on very long time intervals mostly are too large to be usable.

Figure 9 depicts the overall processing time depending on the number of servers. It shows that the mentioned savings even further increase with the number of servers.

The DTI+S scheme achieves these savings with negligible storage consumption compared to the amounts of trajectory data stored by the SP-MOD. With the mentioned  $1.88 \cdot 10^6$  position updates each trajectory consumes about  $1.88 \cdot 10^6 \times 24 \text{ byte} \approx 43 \text{ MB}$ . On the other hand, DTI+S only consumes  $8334 \times 225 \text{ byte} \approx 1.8 \text{ MB}$  per trajectory. Thus, DTI+S accounts for less than 4.2% of the overall storage consumption.

## 8. RELATED WORK

Our research on SP-MODs is related to MODs from the field of database research and scalable location management systems (LMS) from the mobile computing research area.

In the last decade numerous index structures for efficient access to past trajectory data in MODs have been proposed [1, 3, 9–12]. However, none of these index structures addresses distributed database systems like SP-MODs. Therefore they particularly do not consider summaries of segments stored by several servers to reduce routing and processing times. DTI+S is complementary to these approaches. It optimizes the distributed processing including query routing, while these approaches optimize local query processing.

LMS, like the large-scale location services presented in [6] and [17], enable scalable management of MOs' current positions [13]. Similar to our approach, LMS use spatial partitioning for distributing the position data among a set of servers. However, LMS do not store past trajectory data.

PLACE\* is a distributed data stream management system which processes continuous range and k-nearest-neighbor queries on a set of MOs [16]. For scalability, its service area is partitioned to regional servers similar to a SP-MOD. However, it does not process queries on past position data.

In [15] Trajcevski et al present BORA for processing spatio-temporal range queries, i.e. coordinate-based queries. Its system model resembles the system model assumed in this paper. BORA and DTI+S together enable efficient processing of coordinate- and trajectory-based queries.

## 9. CONCLUSIONS

SP-MODs allow for scalable, distributed storage of moving objects' trajectories. An important issue in processing a query  $q$  in a SP-MOD is routing  $q$  to the servers that store the queried data. While the spatial partitioning inherently allows for efficient routing of coordinate-based queries, the efficient routing of trajectory-based queries is challenging.

Therefore, we proposed DTI allowing for efficient routing of trajectory-based queries in SP-MODs. A DTI realizes an overlay network between servers storing segments of a certain trajectory. Our evaluation shows that the DTI scheme significantly reduces the time for routing compared to plain trajectory-based routing. For instance, for a SP-MOD with 1000 servers, DTI reduces the routing time by up to 73%.

Furthermore, we proposed DTI+S, which enhances DTI by maintaining summaries on trajectory segments. The summaries enable efficient processing of queries on aggregates of dynamic attributes like a segment's length, the maximum speed within a segment, or the intersection of a segment and a given region. Our simulations show that DTI+S can reduce the overall processing time by more than 98%.

## 10. ACKNOWLEDGMENTS

The work described in this paper was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center (SFB) 627.

## 11. REFERENCES

- [1] R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.
- [2] Z. J. Haas. The Routing Algorithm for the Reconfigurable Wireless Networks. In *Proc. of 6th ICUPC*, Oct. 1997.
- [3] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB Journal*, 15(2):143–164, June 2006.
- [4] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proc. of 32nd STOC*, 2000.
- [5] A. Leonhardi and K. Rothermel. A Comparison of Protocols for Updating Location Information. *Cluster Computing*, 4(4):355–367, Oct. 2001.
- [6] A. Leonhardi and K. Rothermel. Architecture of a Large-scale Location Service. In *Proc. of 22nd ICDCS*, July 2002.
- [7] M. Liljenstam, J. Liu, and D. M. Nicol. Development of an internet backbone topology for large-scale network simulations. In *Proc. of 2003 Winter Simulation Conf.*, pages 694–702, Dec. 2003.
- [8] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *ACM SIGOPS Review*, 36:131–146, Dec. 2002.
- [9] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull.*, 26(2):40–49, June 2003.
- [10] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Trans. Knowl. Data Eng.*, 19(5):663–678, May 2007.
- [11] M. Pelanis, S. Saltinis, and C. S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Systems*, 31(1):255–298, Mar. 2006.
- [12] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *Proc. of 26th VLDB*, Sept. 2000.
- [13] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE Trans. Knowl. Data Eng.*, 13(4):571–592, July 2001.
- [14] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Comm. ACM*, 33(6):668–676, 1990.
- [15] G. Trajcevski, H. Ding, P. Scheuermann, and I. F. Cruz. BORA: Routing and Aggregation for Distributed Processing of Spatio-Temporal Range Queries. In *Proc. of 8th MDM*, May 2007.
- [16] X. Xiong, H. G. Elmongui, X. Chai, and W. G. Aref. PLACE\*: A Distributed Spatio-temporal Data Stream Management System for Moving Objects. In *Proc. of 8th MDM*, May 2007.
- [17] J. Zhang, G. Zhang, and L. Liu. Geogrid: A scalable location service network. In *Proc. of 27th ICDCS*, June 2007.