Real-Time Range Query Approximation by Means of Adaptive Quad Streaming

Simon Keller and Rainer Mueller
University of Applied Sciences, Constance, Germany

Keywords: Continuous Spatial Range Queries, Mobile Objects, Streaming-based Decentralization, Mobile Environments.

Abstract: Continuous range queries are a common means to handle mobile clients in high-density areas. Most existing approaches focus on settings in which the range queries for location-based services are mostly static whereas the mobile clients in the ranges move. We focus on a category called Dynamic Real-Time Range Queries (DRRQ) assuming that both, clients requested by the query and the inquirers, are mobile. In consequence, the query parameters results continuously change. This leads to two requirements: the ability to deal with an arbitrary high number of mobile nodes (scalability) and the real-time delivery of range query results. In this paper we present the highly decentralized solution Adaptive Quad Streaming (AQS) for the requirements of DRRQs. AQS approximates the query results in favor of a controlled real-time delivery and guaranteed scalability. While prior works commonly optimizes data structures on servers, we use AQS to focus on a highly distributed cell structure without data structures automatically adapting to changing client distributions. Instead of the commonly used request-response approach, we apply a lightweight streaming method in which no bidirectional communication and no storage or maintenance of queries are required at all.

1 INTRODUCTION

Today's ubiquitous mobile devices in everyone's hands and the location sensors they contain lead to both, demand and potential for location-aware services (Kaczor and Kryvinska, 2013). As a consequence of ubiquity, there is an obvious need for more efficiency in *spatial queries* in terms of given QoS parameters. And these parameters certainly depend on the type of the application scenarios considered. There are countless use cases for applications, often in the area of nearby business or leisure which provide mobile users with aggregated information out of spatial queries (e.g. gas stations, hospitals, stores/shopping malls) (Molnár et al., 2014). In contrast to these *stationary* target objects or clients of a system, there are *mobile* ones such as commuters on a busy train looking for free seats, people requiring an ad-hoc ridesharing (cab-hailing) in order to manage the next two blocks, or firefighters in a smoky building, forest or disaster event. In these scenarios the requirement is not to know, who/which are the next k clients (matching the needs) but which clients are within a specific range (*range query*), in our case within in a geometrical range (*spatial range query*) of d_0 meters around a dedicated client. Moreover, not only are the requested clients mobile (e.g. the requested firefighter colleagues), which is a *continuous spatial range query*. Also the requesting clients or inquirers (e.g., a dedicated firefighter looking for colleagues in restricted visibility) are mobile. We call this a *dynamic real-time range query* (DRRQ) in which not only the query results but the queries as such change continuously. The application scenarios behind DRRQs involve a certain degree of unpredictability or spontaneity. We call this class of problems *ad-hoc mobility challenges*, which obviously increases the complexity of continuous range queries, but is much closer to reality nowadays when everyone and everything is mobile. You can easily transfer the described ad-hoc mobility challenges to other domains or objects (firefighters, autonomous vehicle controlling/dispatching, supply chains, business processes, etc.). Many range query approaches are restricted to either static objects or static queries (e.g. (Wu et al., 2004c; Kalashnikov et al., 2002; Xuan et al., 2011)). Others optimize established search algorithms on a centralized single server (e.g. (Mokbel and Aref, 2008; Al-Khalidi et al., 2013; AL-Khalidi et al., 2013)). Even already existing distributed approaches often turn their attention either to the static variants (e.g. (Wang et al., 2006)) or to optimize well-known local data structures such as variants of R or B trees (e.g. (Yu et al., 2019)).

In contrast, we focus on DRRQs and maximize the degree of distribution defining a very fuzzy or foggy edge between mobile clients and servers. Even if all nodes, servers and clients, are involved, their indi-

Keller, S. and Mueller, R.
Real-Time Range Query Approximation by Means of Adaptive Quad Streaming.
DOI: 10.5220/0010246500130024
In *Proceedings of the 10th International Conference on Sensor Networks (SENSORNETS 2021)*, pages 13-24
ISBN: 978-989-758-489-3

vidual local range query processing load is reduced as much as possible. We largely dispense with local range query data structures in favor of using a granular distribution intelligence and a lightweight, fast communication pattern on a streaming basis. The distribution fuzziness results in an approximated, not always completely correct range query result. However, we will show that this inaccuracy can be optimized with respect to real-time support and scalability and is therefore becoming irrelevant.

2 PRECONDITIONS AND REQUIREMENTS

The discussion of two-dimensional DRRQs in the euclidean space and their preconditions can be enhanced by a more formal consideration. In the infrastructure considered, we define $C := \{c^t = (c_{id}, c_1^t, c_2^t)\}$ as the set of mobile client nodes (in the following: *client nodes* or just *nodes*). t of c^t indicates the point in time, at which c is considered. c_{id} is its never changing, system-wide unique identifier. (c_1^t, c_2^t) are the coordinates of c in the two-dimensional euclidean space at point in time t. S is the set of one or more server nodes (in the following: *servers*) and $D := C \cup S$ the distributed system. The nodes in C have the ability to determine their own location (GPS, Bluetooth, Wifi, etc.) and have a network connection (4G/5G, Wifi, etc.) to communicate with the servers in S. For defining a query we use the euclidean norm (or 2-norm) as metric d.
Following these names and notions, a range query can now be specified as a request and response:

Definition 1. *Request $Q(c^t)$ and response $R(c^t)$*
$Q(c^t) :=$ query of c^t at point in time t
$R : C \to \mathcal{P}(C)$ with $R(c^t) := \{b^t \in C \mid d(b^t, c^t) \leq d_0\}$

Following these definitions, we are now able to define different categories of range queries:

Type 1: Static (Spatial) Range Queries
Node locations are constant (immovable clients)
$\Rightarrow R(c^t) = R(c^{t'})$; $\forall t, t' \in \mathbb{R}^+$ with $c^t = c^{t'}$
Queries are constant (immovable inquirers)
$\Rightarrow c^t = c^{t'}$; $\forall t, t' \in \mathbb{R}^+$
Type 2: Continuous (Spatial) Range Queries
Identical to Type 1 except:
Node locations are variable (moving clients)
$\Rightarrow \exists\, t, t' \in \mathbb{R}^+ : c^t = c^{t'} \wedge R(c^t) \neq R(c^{t'})$
Type 3: <u>D</u>ynamic <u>R</u>eal-Time <u>R</u>ange <u>Q</u>ueries
Identical to Type 2 except:
Queries are variable (moving inquirers)
$\Rightarrow \exists\, t, t' \in \mathbb{R}^+ : c^t \neq c^{t'}$

We will discuss related approaches for these query categories in Section 3. Having the considered terms in mind, the requirements needed for a DRRQ solution become straightforward. Under the terms of dealing with high-density systems of millions of nodes per considered area unit (cmp. ITU expectation (International Telecommunication Union ITU, 2015)), every centralized approach (especially a single, centralized or replicated server) obviously becomes a bottleneck. We will discuss this in more detail in Section 4. This leads us to our first requirement:

Requirement A – Scalability: $Q(c^t)$ has to be processed in sublinear time in n, $\forall c^t \in C, \forall t \in \mathbb{R}^+, n = |C|$

In this context, hotspots in which high-density clusters intersect with the range of the query ($\exists c^t \in C : |R(c^t)| \in O(n)$) are of particular interest.

The ad-hoc mobility character of these scenarios (cmp. Section 1) in combination with high-density areas lead to the supply and demand for continuously updated information. To be more precise, the mobility of both requesting and requested nodes (moving clients and inquirers) in DRRQs requires *real-time behavior* in answering the queries as the second requirement for our solution.

Requirement B – Real-time Propagation: $\forall c^t \in C, \forall t \in \mathbb{R}^+, \forall b^{t'} \in R(c^t) : \tau(c^t) - t' \leq t_0$ with $\tau(c^t) :=$ earliest point in time, at which $R(c^t)$ is available to c^t.

t_0 obviously depends on the application domain and can either be a hard or a soft deadline. In a ridesharing scenario a soft deadline of $2000 - 3000ms$ might be acceptable whereas in a firefighter's life-threatening setting a hard deadline of $300ms$ is crucial. As a consequence, recomputing all query results from scratch on every client movement seems to be the wrong approach in contrast to focusing on the deltas.

3 RELATED WORK

Location-dependent queries have been extensively studied in the last decade. While a lot of literature propose general implementations for a spatial query processing (Mokbel et al., 2004; Gedik and Liu, 2004; Tao and Papadias, 2003), the two most dominant research issues related to this paper are range queries (Wu et al., 2004c; Wu et al., 2004a; Wu et al., 2004b; Wu et al., 2005; Kalashnikov et al., 2002) and nearest neighbor queries (Yang and Chiu, 2017; Gao et al., 2014; Wu et al., 2007; Mouratidis et al., 2005a; Mouratidis et al., 2005b) which do not match our research problem directly but provides inspiring solutions. Another related field are spatial alarms

(Bamba et al., 2009), as they use similar techniques to query safe areas. While Continuous Range Queries (moving clients) are often studied (e.g. (Wu et al., 2004c; Wu et al., 2004a; Wu et al., 2004b; Yu et al., 2005; Kalashnikov et al., 2002; Wang et al., 2006)), a much smaller set of literature is found for supporting DRRQs (moving inquirers). We can divide the literature on moving queries into two categories of centralized approaches, such as SINA (Mokbel et al., 2004), SOLE (Mokbel and Aref, 2008), GPAC (Mokbel and Aref, 2005) or (Cheema et al., 2010; Al-Khalidi et al., 2013; AL-Khalidi et al., 2013) and distributed solutions like MobiEyes (Gedik and Liu, 2006) or Distributed Hybrid Index (Yu et al., 2019). The latter meet our Requirement A (cmp. Section 2) of horizontal scalability much better. However, MobiEyes requires a centralized server as a mediator between mobile clients which could obviously lead to the already discussed bottleneck and in turn cannot guarantee free scalability. The Distributed Hybrid Index avoids centralization by providing a predefined grid spread over many servers, but adopts the dynamic character of ad-hoc mobility challenges solely by a local server-internal height-limited variant of an R tree (VR tree). Even if update costs are therefore limited, the tree might degenerate in our high-density scenarios with hotspots in terms of the number of leafs and nodes per leaf. We will come back to this solution later after having described our approach in detail. Another stream in literature on range queries takes assumptions about the objects' movement into account. To optimize query processing, fix road trajectories (Papadias et al., 2003), or motion adaption, indexing (Gedik and Liu, 2004) is used which is not really applicable in an emergency scenario with people in panic whose movements are not predictable. Yet another stream in literature deals with computation offloading in order to optimize the balance between task offloading to expedite the task execution and additional communication costs and latency (Kao et al., 2017; Ying Cai et al., 2006; Gedik and Liu, 2004). This assumption is not always realistic, as clients lack battery capacity and computational or storage capabilities. Indexing techniques for moving objects have been proposed and optimized multiple times in centralized database environments with grid or cell structures (Yu et al., 2005; Zheng et al., 2006; Mouratidis et al., 2009; Šidlauskas et al., 2012; Wang and Zimmermann, 2008), hash-based schemes in buckets (Song and Roussopoulos, 2003), and via tree-based methods (Tayeb, 1998; Prabhakar et al., 2002; Šaltenis and Jensen, 2002; Šaltenis et al., 2000; Jensen et al., 2004). Non-uniformity in mobile object data is also handled by a hierarchical grid index structure (Yu et al., 2005) or a binary tree with spatial operations (Arya et al., 1998). However, the scalability aspect in a distributed mobile environment is not the focus but those works could provide an inspiring base for indexing clients in our distributed solution.

We distinguish our solution from the approaches mentioned above as it provides the following unique properties: (1) Our system can be distributed on an unlimited number of server nodes, fitting an unlimited volume of clients without the need to know the whole system, and (2) we do not separate mobile objects and query consumers but link them and do not assume that there is an imbalance between a large volume of mobile objects and only a few queries. (3) Neither the clients nor the queries are of static nature and our system works well with random movements and non-uniformity. (4) Finally our solution outputs a range query change immediately to the consuming clients to meet real-time requirements.

4 QUAD STREAMING

The Adaptive Quad Streaming (AQS) approach can be described in three parts: (1) the distribution scheme (adaptive cell structure, cmp. 4.1); (2) the interaction pattern (a streaming-based variant, cmp. 4.2); (3) the (range query) processing (algorithms, cmp. 4.3). The scheme applied to the distribution of servers in our approach results from the following consideration: Let us assume that the range query is, as already mentioned in 2, realized straightforwardly through a request-response interaction pattern using one centralized server s_0 for all client nodes. That means, a client joins the system (i.e., becomes a client node $c^t \in C$), later after each movement at time t, it sends a range query $Q(c^t)$ as a *request* to s_0 and receives a result $R(c^t) \subset C$ as a *response* from s_0. Finally, c^t leaves the system. In consequence, s_0 knows all clients that still joined ($c \in C$) and their latest transmitted locations. On each received $Q(c^t)$, s_0 calculates $d(c^t, b^t), \forall b^t \in C$ and, if $d(c^t, b^t) \leq d_0$, sends updated $R(b^t)$ for every previously received $Q(b^t)$ back to node b^t, as well as $R(c^t)$ back to c^t. This brute-force, any-to-any calculation, obviously resulting in complexity $O(n^2)$ with $n = |C|$, cannot fulfill our Requirement A of sublinear processing. There are many approaches in literature addressing a sublinear indexing of mobile nodes in continuous spatial range queries (e.g. (Wu et al., 2004b; Kalashnikov et al., 2002; Yu et al., 2019)). But even in the best case of non-overlapping ranges and constant time for each query

$$R(c^t) = \emptyset \wedge \sigma(Q(c^t)) \in O(1),\ \forall Q(c^t), \forall c^t \in C$$
$$\sigma(O(c^t)) := \text{time span on server side between receiving query } O(c^t) \text{ and sending result } R(c^t)$$

the processing workload for $n \to \infty$ cannot be managed by a single server alone. For an arbitrary $c^t \in C$:

$$\textstyle\sum_{b^t \in C} \sigma(O(b^t)) \in n \cdot O(\sigma(O(c^t))) = O(n)$$

It is always possible to find a greater n to overload all vertically scaled server hardware.

Just the opposite extreme to the server bottleneck would be an approach in which the nodes, as clients, share the range processing uniformly without a centralized component. This is obviously a good precondition for scalability. However, the lack of peer-to-peer communication for coordination and synchronization between clients in a mobile cellular or simply Internet-based network requires a centralized component for mediation. Yet limited resources of mobile nodes in terms of battery inhibit extensive processing on the clients' side. Additionally, limited storage and computation resources of mobile clients might prevent complex calculations of requests on such devices, even on today's hardware (Shao and Taniar, 2014).

As a consequence, we aim at a compromise between both extremes, full server and full client processing. Therefore, a horizontal scaling of server infrastructure in terms of partitioning the processing load between server and client nodes, minimizing the communication load and fitting arbitrary client distributions and counts must be the focus. In order to find an optimal load partitioning for servers and clients we create a self-adapting multidimensional structure built of cells.

4.1 Adaptive Cell Structure

The first step for an adaptive cell structure is the *initial cell* X_0 covering the whole considered two-dimensional area. As all following cells in the structure, this cell is a square. X_0 is assigned to an *initial server* $s_0 \in S$ (see Figure 1).

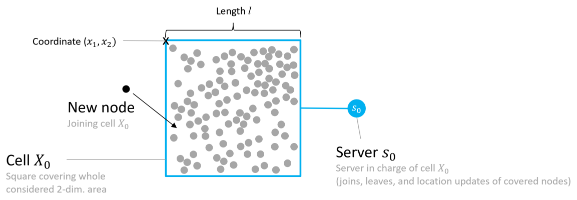

Length l
Coordinate (x_1, x_2)
New node
Joining cell X_0
Cell X_0
Square covering whole considered 2-dim. area
s_0
Server s_0
Server in charge of cell X_0
(joins, leaves, and location updates of covered nodes)

Figure 1: Initial Cell.

If a new node joins the system and therewith the initial cell X_0, the *cellcount* (number of nodes covered by the cell) might exceed a predefined *split threshold*. As a consequence, the initial cell X_0 is divided into four equally-sized *subcells* $X_{(0,1)}, \ldots, X_{(0,4)}$. X_0 is called the *supercell* of $X_{(0,1)}, \ldots, X_{(0,4)}$. For a better generalization we introduce some definitions for these concepts.

A cell $X_u := (x_1, x_2, l) \in \mathbb{R}^3$ is defined by its upper left corner (x_1, x_2) as a coordinate and its edge length l.

XC_u is the set of nodes covered by cell X_u and $|XC_u|$ therefore its cell count.

Definition 2. *Cell node set* XC_u

$$XC_u := \{c = (c_{id}, c_1, c_2) \in C \mid x_1 \le c_1 \le x_1 + l \wedge x_2 \le c_2 \le x_2 + l\}$$

Definition 3. *General indexing notation of cells*

X_u is a cell with index $u = (n_1, \ldots, n_k)$

Subcells of X_u: $X_{v_1}, \ldots, X_{v_4}$ with $v_i = (u, i)$

In the indexing notation of X_u, the $u = (n_1, \ldots, n_k)$ describes its creation history: $u = (0) = 0$ for the initial cell, $\{(0, i) \mid 1 \le i \le 4\}$ for its subcells and so on. This indexing notation is used for understanding only and not in the implementation of the algorithms of the system themselves.

Definition 4. *Thresholds & Relaxed Adaption Factor*

Relaxed Adaption Factor:	$R \in [0, 1]$
Split Threshold:	$C_{split} \in \mathbb{N}$
Merge Threshold:	$C_{merge} := \lfloor R/4 \cdot C_{split} \rfloor$

Using these thresholds, we can now define the split and merge rules:

Split Rule for Cell $X_u = (x_1, x_2, l) : |XC_u| > C_{split} \rightarrow$

1. $X_{(u,i)} := (x_1 + l \cdot i \bmod 2, x_2 + l\lfloor i/2 \rfloor, l/2), 1 \le i \le 4$
2. $XC_{(u,i)} := \{c = (c_{id}, c_1, c_2) \in XC_u \mid x \le c_1 \le x + l \wedge y \le c_2 \le y + l\}, X_{(u,i)} = (x, y, l)$
3. $XC_u := \emptyset$

If the cell count $|XC_u|$ of X_u exceeds the *split threshold* C_{split}, X_u will be split into four new subcells $X_{(u,i)}, 1 \le i \le 4$, each assigned to a new unused server from the pool of servers S (see Figure 2). Note, that the original cell X_u and its server s_u remain for coordination purposes, but do not contain nodes anymore. These nodes were divided between the subcells.

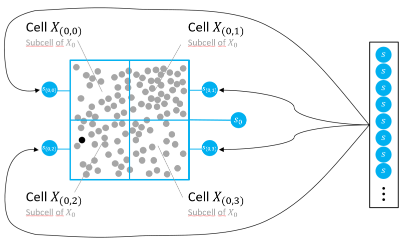


Figure 2: Cell splitting and server assignment.

In the same way, the four cells created during a split are merged again, if their cell counts underrun the *merge threshold* C_{merge}.

Merge Rule for subcells of $X_{(n_1,\ldots,n_k)}$:
$\forall X_{(n_1,\ldots,n_k,i)} : |XC_{(n_1,\ldots,n_k,i)}| < C_{merge} \rightarrow$

1. $XC_{(n_1,\ldots,n_k)} := \cup_{i=1}^{4} XC_{(n_1,\ldots,n_k,i)}$
2. $XC_{(n_1,\ldots,n_k,i)} := \emptyset, 1 \leq i \leq 4$
3. Delete $X_{(n_1,\ldots,n_k,i)}, 1 \leq i \leq 4$

After a merge, the supercell absorbs all nodes of the merged cells. The servers, to which the subcells were assigned, are released and available in the serverpool again. Following this rule, the cell counts of all four cells have to underrun the threshold. But less strict rules might be conceivable, e.g. only two cells are sufficient for a merge.

Before we are able to introduce how this cell structure is used for the DRRQs, we have to define the connections between all nodes, servers and clients. Every cell maintains a maximum of eight links to adjacent cells, three to the neighbor cells created during the split (the other quad cells), one to the supercell, and four to the subcells. Except for the subcell links, all links are mandatory. In addition, every cell has an unsorted simple list with links to the covered clients (see Figure 3). Please, keep in mind, that we would never explicitly set, update, or follow these cell links in the following sections for the sake of clarity. We assume, that these links are just there.

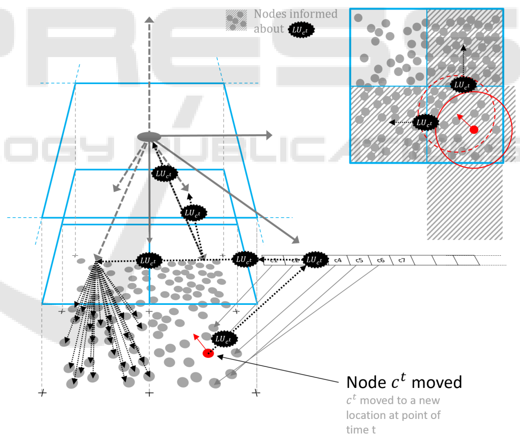
4.2 Streaming

In contrast to most other approaches, we do not use the request-response pattern, neither directly nor implicitly (Yu et al., 2019), for the range queries. Requests for DRRQs are never created, transferred, stored, or updated anywhere in the whole system. Neither is there a need to re-evaluate such a query in case of location updates of covered nodes nor to update a query itself on an inquirer's movement. Instead, we use a streaming communication pattern in which events, as small information chunks, stream from node to node, whether server or client. For instance, location update events permanently flow from moving clients over the cells to all other clients affected by their movements.

Each node of the system, server and client, is able to send arbitrary events to arbitrary receivers using a service primitive *sendEvent*: `sendEvent <receiver address> <event>`. `sendEvent` uses one of the outgoing links from the sending node to the next node specified by `address`. In addition, each node has as service primitive `receiveEvent` forwarding a received event to a node-internal callback registered for this type of event. Every *event* $e := (tp, O_1, O_2, O_3, \ldots)$ is simply a tuple of undefined length, in which $tp \in \mathbb{N}$ is the type of e. O_i are arbitrary objects of any type and represent the information of the event, in which count and type of the objects O_i depend on the type of event tp. Please note that the events themselves include neither a sender nor a receiver address. Therefore, a node receiving an event cannot identify where it originally comes from. In the following, we describe the different event types and their objects:

Definition 5. *Location Update Event* (LU_{c^t})
$e := (0, c^t, t, f)$
$t \in \mathbb{R}$: *timestamp at which* c^t *reached its location*
$c^t \in C$: *node that moved to a new location*
$f \in \mathbb{N}$: *flag (0: regular, 1: join, 2: connected)*

The location update events (in the following often simply LU_{c^t}) are used to inform that a node c_t moved at point in time t to a new location. The new location is stored in the node itself (cmp. Section 2). This is the most important event type in the system. The events of this type are initially created by the nodes themselves after they have moved to a new location (see Figure 3). The flag f of LU_{c^t} is used to transmit a status intended by the event and is on demand changed by nodes while flowing over them. A *join* status intends that the node specified in the event is either new to the system and wants to join it (*system join*) or has left a cell and wants to join another cell (*cell join*). Later on, we will see that the differentiation between the two *join* types is not relevant. We define c^t as *Connected Node*, if it is connected to s_u in the sense of, it is covered by the cell X_u assigned to s_u. As we know from Subsection 4.1, every server uses an unsorted list of links to all its connected nodes. Note, that a connected node itself also maintains a link (or a *connection*) to its server. The link between a con-



Nodes informed about LU_c
LU_c
c4
c5
c6
c7
Node c^t moved
c^t moved to a new location at point of time t

Figure 3: Location update event streaming.

nected node and its server is therefore always bidirectional. A *connected* flag in an LU_{c^t} indicates, that the node specified in the event is already connected to the server that received the event. A *regular* flag tells the receiving node that the client node c^t specified in the LU_{c^t} is neither connected to this server nor wants to join it.

Definition 6. *Connected Event (CON_{s_u})*
$e := (1, s_u)$
$s_u \in S$: *server the receiver is now connected to*

A node receives a CON_{s_u} event if it has been connected to (new) server s_u. As a consequence, only client nodes receive this type of event which happens for one of the following three reasons:

1. New node joins: A new node c^t wants to join the system and therefore already sent an LU_{c^t} with the join flag to a node of the system D. After having received CON_{s_u} the node knows that it is connected to D for the first time over server s_u.
2. Node moved to non-covered location: A node c^t moved to a location where it lost cell coverage by its original connected cell and therefore joined another server s_u whose cell X_u covers the node.
3. Cell split: After a cell has been split, the connected nodes are divided between the four sub cells. This is also realized by a LU_{c^t} with join flag which in turn results in a CON_{s_u} sent by the individual subcells.

Definition 7. *Merge Request Event ($MREQ_{X_u}$)*
$e := (2, X_u)$
$X_u \in X$: *cell X_u whose server s_u wants to merge*

Definition 8. *Merge Acknowledged Event (MACK)*
$e := (3)$

If a cell count underruns the merge threshold C_{merge}, the server s_u of the corresponding cell X_u sends a MREQ to the server of the supercell. If a cell received MREQs from all four subcells, it initiates a merge action and in turn sends a MACK to the servers of the four subcells to request their closing (cmp. merge rule in Subsection 4.1).

We will now explain how these streamed events and the cell structure described above are used in the algorithms of the system D.

4.3 Algorithms

In the following, the algorithmic parts of the system D and our *Adaptive Quad Streaming* approach are described in more detail from the servers' perspective. However, we start with a client activity since this is the most straightforward way to introduce the server algorithms consistently. Following our streaming interaction approach, most algorithms are realized as callbacks receiving and sending events.

If a new (client) node c^t wants to join the system, it sends a location update event with a join flag to the server s_0 of the initial cell X_0 (cmp. 4.1) using its `sendEvent` service primitive. However, s_0 handles this initial event just like any other event of this type. From s_0's point of view, it does not matter whether the node c^t specified in the event is new to the system or just new to its cell. The only interesting aspect for s_0 is that c^t wants to join and whether its cell X_0 covers c^t. It will then be connected to s_0. Otherwise, s_0 passes this event to the next adjacent cell in direction to c^t's location. And this is the uniform behavior of every server. The only difference is that the path (number of hops) from s_0 to the server the node is later connected to might be longer compared to a usual node's movement. The reason is that s_0 is the only entry point into the system regardless of the node's initial position.

Figure 4 describes the central algorithm in terms of a callback function (`locationUpdateReceived`) handling the scenario mentioned above and all other cases, in which server nodes received a location update event. In this algorithm, the server $s_{(n_1,\ldots,n_k)}$ first checks the flags of the passed location update event $LU_{c^t} := (0, c^t, t, f)$. If it is not flagged, i.e. a regular event, than c^t neither wants to join nor is connected to $s_{(n_1,\ldots,n_k)}$ (line 2). In this case, the only remaining task for $s_{(n_1,\ldots,n_k)}$ is to check, if the node's range intersects with its cell X_0 and if so, to inform all connected nodes $b \in XC_{(n_1,\ldots,n_k)}$ about the location update event LU_{c^t} (lines 2–4). In case of an LU_{c^t} flagged as *join*, $s_{(n_1,\ldots,n_k)}$ checks if its cell $X_{(n_1,\ldots,n_k)}$ covers c^t. If yes and if there are no further subcells potentially covering this node (lines 5–6), then c^t will be connected to $s_{(n_1,\ldots,n_k)}$. Therefore, $s_{(n_1,\ldots,n_k)}$ adds c^t to its unsorted list, checks if its cell has now to split (cmp. Figure 5) and informs c^t about now being connected to $s_{(n_1,\ldots,n_k)}$ (lines 7–9). Before informing all other connected nodes about c^t, LU_{c^t} is reflagged with 0, since the join of c^t is already executed (lines 10–12). In case of LU_{c^t} being flagged as *connected*, $s_{(n_1,\ldots,n_k)}$ checks if c^t is still covered by its cell after the movement (lines 13–14). If yes, the LU_{c^t} is reflagged with 0 before informing all other nodes, since no further reconnection of c^t is required (lines 15–17). If no, c^t is disconnected from $s_{(n_1,\ldots,n_k)}$ and therefore removed from the unsorted list of nodes (line 19). In case of underrunning the merge threshold, $s_{(n_1,\ldots,n_k)}$ requests a merge from its supercell server $s_{(n_1,\ldots,n_{k-1})}$ (lines 20–21). After disconnecting c^t from $s_{(n_1,\ldots,n_k)}$, c^t has to join another cell. Therefore, the LU_{c^t} is reflagged with

1 before informing all other connected nodes about this news (lines 22–25). All server- or cell-internal work is now done. The following lines are for passing the event to the adjacent neighbors, if they exist and their cells intersect with c^t's range: (1) subcells (lines 26–29); (2) the three quad neighbor cells (lines 30–32); (3) the supercell, if c^t's range covers the supercell completely (lines 33–34). Please note, that

Data:

```
    $s_{(n_1,...,n_k)}$ : server that received a $LU_{c^t}$
    $X_{(n_1,...,n_k)}$ : cell assigned to $s_{(n_1,...,n_k)}$
    $X_{(n_1,...,n_k,i)}$ : potential subcells of $X_{(n_1,...,n_k)}$
    $X_{(n_1,...,n_{k-1})}$ : supercell of $X_{(n_1,...,n_k)}$

 1: function LOCATIONUPDATERECEIVED($c^t \in C, t \in \mathbb{R}, f \in \mathbb{N}$)
 2:     if $f = 0 \wedge$ rangeIsIntersected($X_{(n_1,...,n_k)}, c^t$) then
 3:         for all $b \in XC_{(n_1,...,n_k)}$ do
 4:             sendEvent($b, LU_{c^t}$)
 5:     if $f = 1 \wedge$ nodeIsCovered($X_{(n_1,...,n_k)}, c^t$) then
 6:         if $\neg$ exists($X_{(n_1,...,n_k,i)}$) then
 7:             $XC_{(n_1,...,n_k)} := XC_{(n_1,...,n_k)} \cup \{c^t\}$
 8:             splitCheck($X_{(n_1,...,n_k)}$)
 9:             sendEvent($c^t, CON_{s_{(n_1,...,n_k)}}$)
10:             f:=0
11:             for all $b \in XC_{(n_1,...,n_k)} \setminus \{c^t\}$ do
12:                 sendEvent($b, LU_{c^t}$)
13:     if $f = 2$ then
14:         if nodeIsCovered($X_{(n_1,...,n_k)}, c^t$) then
15:             f:=0
16:             for all $b \in XC_{(n_1,...,n_k)} \setminus \{c^t\}$ do
17:                 sendEvent($b, LU_{c^t}$)
18:         else
19:             $XC_{(n_1,...,n_k)} := XC_{(n_1,...,n_k)} \setminus \{c^t\}$
20:             if $|XC_{(n_1,...,n_k)}| < c_{merge}$ then
21:                 sendEvent($s_{(n_1,...,n_{k-1})}, MREQ_{X_{(n_1,...,n_k)}}$)
22:             f:=1
23:             if rangeIsIntersected($X_{(n_1,...,n_k)}, c^t$) then
24:                 for all $b \in XC_{(n_1,...,n_k)} \setminus \{c^t\}$ do
25:                     sendEvent($b, LU_{c^t}$)
26:     if exists($X_{(n_1,...,n_k,i)}$) then
27:         for all $i \in \{1,...,4\}$ do
28:             if rangeIsIntersected($X_{(n_1,...,n_k,i)}, c^t$) then
29:                 sendEvent($s_{(n_1,...,n_k,i)}, LU_{c^t}$)
30:     for all $j \in \{1,...,4\} \neq n_k$ do
31:         if rangeIsIntersected($X_{(n_1,...,n_{k-1},j)}, c^t$) then
32:             sendEvent($s_{(n_1,...,n_{k-1},j)}, LU_{c^t}$)
33:     if $\neg$ rangeIsCovered($X_{(n_1,...,n_{k-1})}, c^t$) then
34:         sendEvent($s_{(n_1,...,n_{k-1})}, LU_{c^t}$)
```

Figure 4: Callback processing an LU_{c^t} event.

the functions `rangeIsIntersected`, `rangeIsCovered`, and `nodeIsCovered` are not described in detail, since these are simple functions with the given coordinates of cells and nodes.

Figure 5 describes how the split threshold is verified in function `splitCheck` (line 2) and how the split of the cell is performed, if required. As a consequence, the subcells for the split are created and the required servers allocated from the server pool to which the subcells are assigned (lines 3–6). Afterwards, all connected nodes are divided between the subcells by sending a join-flagged location update event to this subcell that covers the respective node (lines 7–11). Keep in mind, that a splitted cell has no connected nodes anymore.

Data:

```
    $s_{(n_1,...,n_k)}$ : server that received a $LU_{c^t}$
    $X_{(n_1,...,n_k)}$ : cell assigned to $s_{(n_1,...,n_k)}$
    $X_{(n_1,...,n_k,i)}$ : potential subcells of $X_{(n_1,...,n_k)}$

 1: function SPLITCHECK(X $X_{(n_1,...,n_k)}$)
 2:     if $|XC_{(n_1,...,n_k)}| > C_{split}$ then
 3:         for all $i \in \{1,...,4\}$ do
 4:             createCell($X_{(n_1,...,n_k,i)}$)
 5:             $s_{(n_1,...,n_k,i)}$ :=allocateServer()
 6:             assignCellToServer($s_{(n_1,...,n_k,i)}, X_{(n_1,...,n_k,i)}$)
 7:         for all $b \in XC_{(n_1,...,n_k)}$ do
 8:             $LU_b := (0, b, t, 1)$
 9:             for all $i \in \{1,...,4\}$ do
10:                 if nodeIsCovered($X_{(n_1,...,n_k,i)}, c^t$) then
11:                     sendEvent($s_{(n_1,...,n_k,i)}, LU_b$)
12:         $XC_{(n_1,...,n_k)} := \emptyset$
```

Figure 5: Split check for a cell.

In the callback `mergeRequestReceived`, the server $s_{(n_1,...,n_{k-1})}$ received a merge request event *MREQ* from subcell server $s_{(n_1,...,n_k)}$. If the number of MREQs received from different subcells in total exceeds the given limit (default is 4, i.e. all four subcells need to request for a merge), then the merge is executed. For that purpose $s_{(n_1,...,n_{k-1})}$ sends a merge-acknowledged event (MACK) to the subcell servers.

The server $s_{(n_1,...,n_k)}$ received the MACK event in callback `mergeAckReceived`. In turn, it passes all connected nodes back to the server $s_{(n_1,...,n_{k-1})}$ of the supercell by sending join-flagged location update events for them. Afterwards, it releases itself back to the server pool. The callbacks `mergeRequestReceived` and `mergeAckReceived` are not described in pseudocode syntax due to space restrictions. Please note, that all event receivers has to ensure, that events are not directly and redundantly sent back to its receivers (*rebound effect*). This can simply be realized by adding flags or sender addresses to the send and received primitives. We have skipped these rebound parts in the algorithms for the sake of clarity.

Finally, if a (client) node b^t receives a location update event $LU_{c^t} := (0, c^t, t, 0)$, it has to check if c^t is in its range: $d(b^t, c^t) \leq d_0$. If yes, b^t can update its local data structures accordingly. If no, b^t has to drop c^t. In this case, b^t was falsely informed by a location update event and uselessly spent limited local resources for this check. The question is how often this happens in terms of probability. We will come back to this in Section 6.

5 EVALUATION

The major benefit of AQS is the very symmetric and regular self-adapting structure resulting in several advantages. Compared to other approaches handling the dynamic behavior of DRRQs by a more or less static distribution plus a flexible server-internal data structure, AQS does not need complex server data structures and handles the dynamics by using a very flexible distribution pattern. Compared to DHI (Yu et al., 2019) for example, we use a simple unsorted list and an self-adapting cell structure instead of an local R tree variants and a fixed distribution grid. In contrast to DHI, the number of nodes per server (cell count) and the branching factor (number of network links to cell neighbors, sub- and supercells) are limited in AQS. The update costs for a split or merge are therefore constant and do not influence cells beyond the direct links. Keeping the supercell saves update costs for neighbor links. The communication load can be limited due to the very regular cell structures: before sending an event to an adjacent cell, a cell can easily check whether the other one covers the node or its range due to its known coordinates and dimensions. Instead of parallelizing the calculation and maintenance of saved range queries on a streaming basis, as with DHI (Yu et al., 2019), we completely renounce the storage of range queries and use streaming in its originally intended sense as a flowing event data stream from clients to clients. Once, an location update event from a client reaches the cell level, it spreads in a waveform-like manner through the cell structure until the cells transfer it back to their connected nodes affected by the update.

Scalability. The scalability is given by construction of the adaptive quad cell structure: in the best possible case of uniformly distributed nodes, the cell structure splits very regularly with new nodes (system join). If one cell X_u (of capacity $m = C_{split}$) splits into subcells $X_{(u,i)}$, the path length for the covered nodes to other nodes might increase by one due to the remaining supercell X_u. But at the same time, $\cup X_{(u,i)}$ provides free capacity for covered nodes of $3m$ until the next split due to the uniform distribution. In total, $\cup X_{(u,i)}$ cover $4m$ nodes and after further k splits, $\cup X_{(u,i,n_1,\dots,n_k)}$ cover $4^k m$ nodes. As a consequence, for $n \to \infty$ the cell count (server load) stays limited and the path length between two client nodes c^t and b^t and therewith the update time of a $Q(c^t)$ increases in a logarithmic manner (base 4), if we use a limited average latency value per hop on the path. This assumption also holds for non-uniformly distributed scenarios since the average cell size decreases exponentially (by splits) in high-density areas whereas the path length grows logarithmically only. That means that in these areas, the logarithmic behavior holds, whereas in other areas with fewer new nodes, the path length remains equal or grows in a sublogarithmic manner. The exponentially decreasing cell size leads to another result.

Approximation. The cell structure approximates the spatial distribution of the client nodes in terms of cell granularity. In high-density regions (hotspots), we have much smaller cells which gradually adapt, i.e. the nearer the hotspot edges get, the larger the cell size will be. Much more interesting is the fact that this way the cell structure approximates the ranges of the queries in terms of falsely informed clients. The better our cell structure approximates the shape of the ranges on average, i.e. the circle lines, the less clients are falsely informed. In order to motivate this assumption, consider the fact that our system leads to a coarsely granulated cell structure in low-density areas. As a result, the cells approximate the shape (circle line) of ranges less exactly compared to high-density areas. But the influence of these coarse bigger cells is reduced, since one of these cells cover fewer nodes compared to areas of this size (covered by several cells) in high-density regions. Therefore, even if big cells inaccurately approximate the circle line only, the number of falsely informed clients is still small with respect to C_{split}. If the circle line intersects a cell, whether large or small, there is always only a limited number of falsely informed nodes in it. As a result, with an optimized pair of (C_{split}, C_{merge}), the range can be approximated as well as possible in order to minimize the falsely informed clients. We will demonstrate this in our experimental results in the following section.

6 EXPERIMENTAL RESULTS

For our performance test we consciously rely on very limited hardware in terms of resources (single-board computers Raspberry Pi 3 B+ with a 1.40 GHz quad-core ARM processor and 1 GB main memory). As OS we choose a minimal Unix distribution to reduce external influence and run our Java implementation on the Java Virtual Machine. To test different client

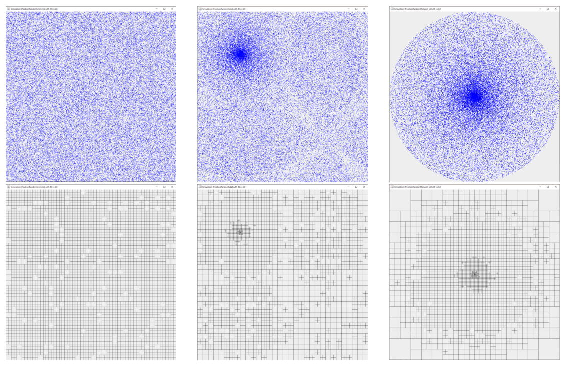

(a) Uniform (b) Unequal (c) Hotspot

Figure 6: Tested client distribution scenarios and resulting cell structure.

distributions with non-uniformity in high- and low-density areas, we create three deterministic scenarios of client movements the positions of which are visualized as snapshots in Figure 6. To achieve comparable results, we always test with a uniform client distribution (Figure 6a) against a very high-density hotspot with empty areas on the edge (Figure 6c) and a less unequal distribution with a non-centered hotspot (Figure 6b). With an increasing probability

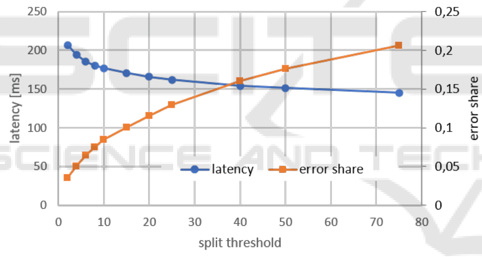


Figure 7: Split threshold influence on error share and latency.

of the nodes moving back to their spawn point the further they move away, we ensure, that the distribution pattern does not change significantly with random client movement over time. A resulting cell structure created by AQS is also displayed in Figure 6 for each of the example distribution scenarios ($C_{split} = 40$). Obviously, the client distributions are represented very well by the cell structure in the system. Different movement velocities are considered to get more realistic scenarios. The velocities allow each client to move randomly, limited to a max. step size of 5% of the whole simulation area's width or height per update. We run our tests with $|C| = 100$'000 clients in an area of 1000×1000 and a query range of $d_0 = 50$. To achieve comparable simulation results, all clients update their location in the same time interval, and after 100'000 updates a system snapshot is recorded for the result sets described in the following.

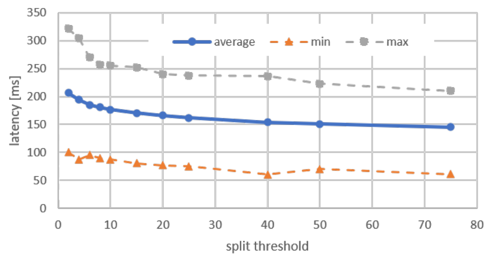


Figure 8: Min/max latency in hotspot scenario.

Effect of Split Threshold C_{split}. As already mentioned in Section 4.3, the approximation of a range query leads to falsely sent LU_{c^t} messages received by a client node b^t. Filtering those messages unnecessarily consumes limited client resources. However, to reduce this proportion named *error share*, the threshold C_{split} can be decreased which leads to a better approximation of the range query as demonstrated in Figure 7 used with our hotspot scenario. At the same time, the communication effort increases as more cells must be addressed while processing a location update with a smaller C_{split} in a deeper cell structure. This is also shown in Figure 7 as *latency* representing an average time between sending an LU_{c^t} from the updating client node c^t to all receiving client nodes b^t. Depicting additionally the minimum and maximum measured latency for all LU_{c^t}s, Figure 8 shows that there is no measured latency which strongly deviates from the mean value even in a hotspot scenario. However, in most real-life scenarios a real-time deadline t_0 of 300ms as in the firefighters scenario could easily be fulfilled with the measured latency in our simulation results, even in worst case.

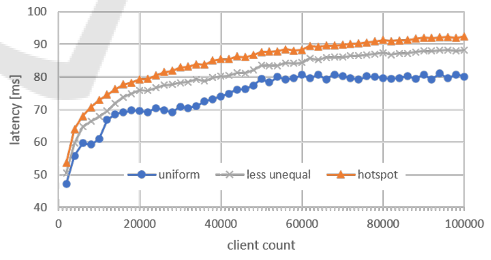


Figure 9: Scalability with increasing client count.

Scalability with Increasing Client Count. To test robustness in scalability on high-density clusters of mobile clients, we record the average latency of LU_{c^t} while spawning an increasing count of clients (system join). This is shown in Figure 9 for our different distribution scenarios ($C_{split} = 40$). As we can see, our solution performs sublinear with respect to an increasing client count and is not much affected by a non-uniform client distribution. Nevertheless we discuss the influence of client distribution in the next paragraph.

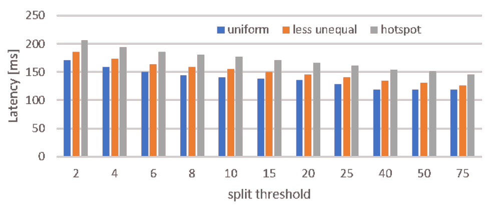


Figure 10: Distribution effect on latency.

Effect of Hotspots. Hotspots are simulated with our distribution scenarios described above and visualized in Figure 6. As a high-density cluster of clients yield to a deeper cell structure and a higher number of cells in those areas, the average latency of an LU_{c^t} event increases more slowly as the error share decreases. This can be easily identified by comparing Figure 10 and Figure 11: the first one shows the increasing latency for different split sizes and distribution scenarios whereas the latter visualizes the error share in the same way.
Reducing the error share in a high-density area is a highly desirable result, as neighbor links in a fix query range rise very sharply in high-density hotspots and challenge client nodes in maintaining many connections.

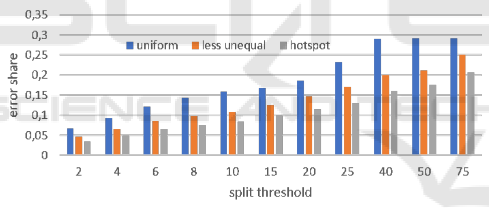


Figure 11: Distribution effect on error share.

Throughput in a Cell. While we are dealing with limited server hardware, the throughput of events handled by a cell within a certain time span is an interesting point. As demonstrated in Figure 12, we could record an average throughput of $\approx 10/ms$ in an in-out-measuring setting. The vertical axis shows the average amount of events which can be processed by a cell in one single millisecond whereas the horizontal axis displays the simulation breakpoints recorded by us over time while our clients are moving and producing a variety of different events. In Figure 13 we show the workload of cell generated by the events (first is uniform, second is unequal, third is hotspot distribution). We use a uniform client distribution of 100‘000 LU_{c^t}s of 100‘000 clients. They are nearly equally spread over all cells regardless of their structure layer. As we can see, leaf cells handle fewer events compared to other cells as expected due to missing subcells. However, that is advantageous as the leaf cells

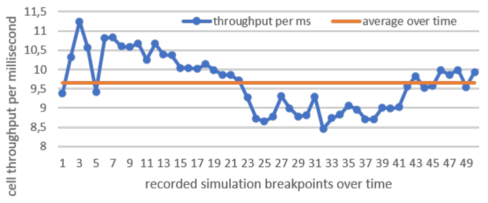


Figure 12: Throughput per cell over time.

have to handle their connected client nodes. Maintaining these communication connections consumes additional calculation power in mobile environments (e.g. availability checks of unreliable mobile devices, network changes, etc.).

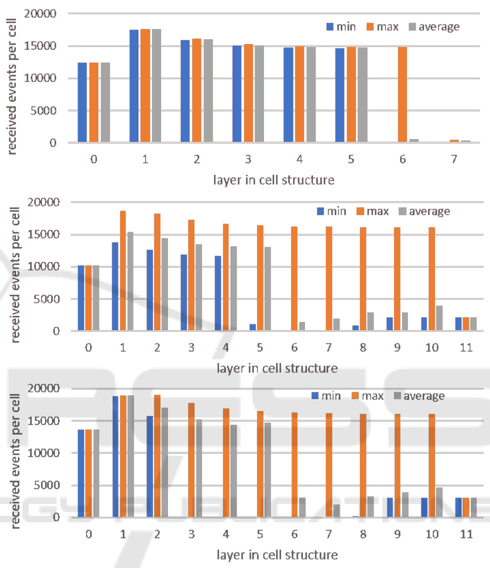


Figure 13: Received events per cell per layer (uniform, unequal, hotspot).

An even more interesting point is the event workload in non-uniform distributions displayed in Figure 13. We identify the same general behavior of an equally spread workload in upper cell layers as with uniform distributions. But in lower cell structure layers the workload drops dramatically on average. An explanation could be the many low-density areas in those scenarios which are near a hotspot and therefore splitted in a deeper layer. The important measured value is the maximum workload received by a cell which stays equally regardless of the distribution pattern and indicates that our system does not overload any cell as expected.

Effect of Client Velocity. In Figure 14, we evaluate the influence of client movement velocity on latency. The horizontal axis represents the client's velocity in random movements which is realized with the step size displayed on the axis, but in a random angle. Although more cells are involved in high velocity up-

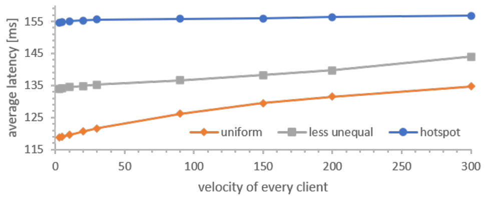


Figure 14: Influence of velocity on latency.

dates, the count of cells receiving an LU_{c^t} (describing a movement in time from $t-1$ to t) is limited. In the worst case, the earlier query result $R(c^{t-1})$ and new one $R(c^t)$ do not intersect at all. This results in a processing of all cells which are covered by both queries. As also shown in Figure 14, AQS is robust enough to deal with arbitrary client velocities as it hardly has any influence on the end-to-end latency.

7 CONCLUSION AND FUTURE WORK

In this paper we focused on a subclass of continuous spatial range queries, the *Dynamic Real-Time Range Queries (DRRQ)*. In these queries, both, requesting and requested nodes, are mobile, leading to always changing queries and results. The DRRQs address a category of mobile application scenarios we call ad-hoc mobility challenges. They are rapidly gaining importance nowadays as clients in high-density systems demand ad-hoc information (movement) updates about other clients nearby in a unpredictable and therefore real-time manner. In order to address this scalability and real-time requirements, we presented *Adaptive Quad Streaming (AQS)* as a highly distributed approach for DRRQs. It combines a dynamically self-adapting cell structure with a lightweight streaming approach. Along with a cell-per-server assignment it adapts to arbitrary and always changing clients distributions using a splitting and merging method and approximates the results of DRRQs. We have shown in simulations that the error rate of the approximation decreases more rapidly than the end-to-end latency (one client's movement update to another client) increases with respect to the used threshold for splitting. With a latency of 150ms we achieved an error rate of 16% whereas an acceptable latency of 200ms lowers the error rate to 4.5%. This is obviously a good quality measure for the approximation of client distributions and corresponding range shapes of client queries. AQS strictly limits the number of clients handled by a cell and server as well as the number of links to other servers, that way controlling the processing and communication load. In consequence, we also showed in our experiments, that the latency grows logarithmically in the number of nodes in the system. This is an indicator for the scalability of AQS, capable to deal with arbitrary high numbers of clients per area unit. The approximation of client distributions by dividing the work and communication load of DRRQs between several servers and (in a very limited way) also clients can be seen and also visualized as a fuzzy two-dimensional edge between clients and servers. This might be an interesting theoretical insight in dynamically adapting edge computing.

Our future work lies in finding out how to assign a server's free capacity more flexibly instead of assuming an unlimited server pool and a one-server-per-cell assignment in an idealistic manner. Another perspective might be how to react on imbalances between processing and communication load as a degeneration effect among the servers arising over time. A challenging aspect might be the influence to the results after a shift to more than two and also non-spatial client dimensions, thus representing very realistic and considerably more complex application scenarios.

REFERENCES

AL-Khalidi, H., Taniar, D., Betts, J., and Alamri, S. (2013). On finding safe regions for moving range queries. *Mathematical and Computer Modelling*.

Al-Khalidi, H., Taniar, D., and Safar, M. (2013). Approximate algorithms for static and continuous range queries in mobile navigation. *Computing*.

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y. (1998). An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*.

Bamba, B., Liu, L., Iyengar, A., and Yu, P. S. (2009). Distributed processing of spatial alarms: A safe region-based approach. *International Conference on Distributed Computing Systems*.

Cheema, M. A., Brankovic, L., Lin, X., Zhang, W., and Wang, W. (2010). Multi-guarded safe zone: An effective technique to monitor moving circular range queries. *International Conf. on Data Engineering*.

Gao, S., Ji, C., Xu, C., and Yang, N. (2014). Parallel spatial nearest neighbour query based on grid index. *2014 IEEE Workshop on Electronics, Computer and Applications, IWECA 2014*.

Gedik, B. and Liu, L. (2004). MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *Advances in Database Technology - EDBT 2004*. Springer Berlin Heidelberg.

Gedik, B. and Liu, L. (2006). MobiEyes: A distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing*.

International Telecommunication Union ITU (2015). IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond.

Jensen, C. S., Lin, D., and Ooi, B. C. (2004). +Query and Update Efficient B-Tree Based Indexing of Moving Objects. In *Proceedings 2004 VLDB Conf.* Elsevier.

Kaczor, S. and Kryvinska, N. (2013). It is all about services-fundamentals, drivers, and business models. *Journal of Service Science Research*.

Kalashnikov, D., Prabhakar, S., Hambrusch, S., and Aref, W. (2002). Efficient evaluation of continuous range queries on moving objects. *Lecture Notes in Computer Science*.

Kao, Y. H., Krishnamachari, B., Ra, M. R., and Bai, F. (2017). HERMES: Latency Optimal Task Assignment for Resource-constrained Mobile Computing. *IEEE Transactions on Mobile Computing*.

Mokbel, M. F. and Aref, W. G. (2005). GPAC: Generic and progressive processing of mobile queries over mobile data. *Sixth International Conference on Mobile Data Management, MDM'05*.

Mokbel, M. F. and Aref, W. G. (2008). SOLE: Scalable on-line execution of continuous queries on spatio-temporal data streams. *VLDB Journal*.

Mokbel, M. F., Xiong, X., and Aref, W. G. (2004). SINA. In *The 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*. ACM Press.

Molnár, E., Molnár, R., Kryvinska, N., and Greguš, M. (2014). Web intelligence in practice. *Journal of Service Science Research*.

Mouratidis, K., Bakiras, S., and Papadias, D. (2009). Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE Transactions on Mobile Computing*.

Mouratidis, K., Hadjieleftheriou, M., and Papadias, D. (2005a). Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. *ACM SIGMOD Int. Conference on Management of Data*.

Mouratidis, K., Papadias, D., Bakiras, S., and Tao, Y. (2005b). A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Transactions on Knowledge and Data Engineering*.

Papadias, D., Mamoulis, N., Zhang, J., and Tao, Y. (2003). Query processing in spatial network databases. *29th International Conference on Very Large Data Bases*.

Prabhakar, S., Xia, Y., Kalashnikov, D. V., Aref, W. G., and Hambrusch, S. E. (2002). Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*.

Šaltenis, S. and Jensen, C. S. (2002). Indexing of moving objects for location-based services. *International Conference on Data Engineering*.

Šaltenis, S., Jensen, C. S., Leutenegger, S. T., and Lopez, M. A. (2000). Indexing the positions of continuously moving objects. *SIGMOD Record (ACM Special Interest Group on Management of Data)*.

Shao, Z. and Taniar, D. (2014). Range-based nearest neighbour search in a mobile environment. *12th International Conference on Advances in Mobile Computing and Multimedia, MoMM 2014*.

Šidlauskas, D., Šaltenis, S., and Jensen, C. S. (2012). Parallel main-memory indexing for moving-object query and update workloads. *The ACM SIGMOD International Conference on Management of Data*.

Song, Z. and Roussopoulos, N. (2003). SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management*. Springer Berlin Heidelberg.

Tao, Y. and Papadias, D. (2003). Spatial Queries in Dynamic Environments. *ACM Transactions on Database Systems*.

Tayeb, J. (1998). A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*.

Wang, H. and Zimmermann, R. (2008). Snapshot location-based query processing on moving objects in road networks. *GIS: ACM International Symposium on Advances in Geographic Information Systems*.

Wang, H., Zimmermann, R., and Ku, W.-S. (2006). Distributed Continuous Range Query Processing on Moving Objects. In *Database and Expert Systems Applications*. Springer Berlin Heidelberg.

Wu, K. L., Chen, S. K., and Yu, P. S. (2004a). Indexing continual range queries for location-aware mobile services. *2004 IEEE International Conference on e-Technology, e-Commerce and e-Service, EEE 2004*.

Wu, K. L., Chen, S. K., and Yu, P. S. (2004b). Indexing continual range queries with covering tiles for fast locating of moving objects. *International Conference on Distributed Computing Systems*.

Wu, K. L., Chen, S. K., and Yu, P. S. (2004c). Processing continual range queries over moving objects using VCR-based query indexes. *MOBIQUITOUS 2004 - 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*.

Wu, K. L., Chen, S. K., and Yu, P. S. (2005). On incremental processing of continual range queries for location-aware services and applications. *MobiQuitous 2005: Second Annual International Conference on Mobile and Ubiquitous Systems -Networking and Services*.

Wu, W., Guo, W., and Tan, K. L. (2007). Distributed processing of moving K-nearest-neighbor query on moving objects. *Int. Conference on Data Engineering*.

Xuan, K., Zhao, G., Taniar, D., Rahayu, W., Safar, M., and Srinivasan, B. (2011). Voronoi-based range and continuous range query processing in mobile databases. *Journal of Computer and System Sciences*.

Yang, K.-T. and Chiu, G.-M. (2017). Monitoring continuous all -nearest neighbor query in mobile network environments. *Pervasive and Mobile Computing*.

Ying Cai, Hua, K., Guohong Cao, and Xu, T. (2006). Real-time processing of range-monitoring queries in heterogeneous mobile databases. *IEEE Transactions on Mobile Computing*.

Yu, X., Pu, K. Q., and Koudas, N. (2005). Monitoring k-nearest neighbor queries over moving objects. *International Conference on Data Engineering*.

Yu, Z., Xhafa, F., Chen, Y., and Ma, K. (2019). A distributed hybrid index for processing continuous range queries over moving objects. *Soft Computing*.

Zheng, B., Xu, J., Lee, W.-C., and Lee, D. L. (2006). Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services. *The VLDB Journal*.