# DRAGON: Multidimensional Range Queries on Distributed Aggregation Trees

Emanuele Carlini[b], Alessandro Lulli[a,b], Laura Ricci[a,b]

[a]*Department of Computer Science, University of Pisa*
[b]*Information Science and Technologies Institute, ISTI-CNR, Pisa*

**Abstract**

Distributed query processing is of paramount importance in next-generation distribution services, such as Internet of Things (IoT) and cyber-physical systems. Even if several multi-attribute range queries supports have been proposed for peer-to-peer systems, these solutions must be rethought to fully meet the requirements of new computational paradigms for IoT, like fog computing. This paper proposes DRAGON, an efficient support for distributed multi-dimensional range query processing targeting efficient query resolution on highly dynamic data. In DRAGON nodes at the edges of the network collect and publish multi-dimensional data. The nodes collectively manage an aggregation tree storing data digests which are then exploited, when resolving queries, to prune the sub-trees containing few or no relevant matches. Multi-attribute queries are managed by linearising the attribute space through space filling curves. We extensively analysed different aggregation and query resolution strategies in a wide spectrum of experimental set-ups. We show that DRAGON manages efficiently fast changing data values. Further, we show that DRAGON resolves queries by contacting a lower number of nodes when compared to a similar approach in the state of the art.

## 1. Introduction

An efficient, scalable and adaptable discovery service is essential for widely distributed services and infrastructures, such as cyber-physical networks, smart-cities platforms and peer-to-peer networks. These services can be grouped under the broader definition of the Internet of Things (IoT) [1], which requires the organization of large-scale infrastructures of smart-"things", such as smartphones and RFID tags. In this context, an important building block for IoT infrastructures is the ability to sustain widely distributed, dynamic, and autonomic services [2].

Fog computing [3], has been recently proposed by Cisco both as computational and architectural paradigm for IoT. Fog computing requires to run services throughout the network, including in specialized routers and in dedicated computing nodes. The result is that intelligence is not localized on centralized cloud computing nodes, but spread throughout in the network. Real time low-latency services may be performed at the edges of the network close to users, while latency-tolerant tasks can be efficiently performed on powerful resources in the core of the network. This new computational paradigm is naturally supported by a hierarchical multi-tier architecture which exploits increasing levels of data aggregation when moving from the edges to the core of the network.

Several services previously developed for highly distributed platforms like peer-to-peer networks are also required in the fog. A support for the resolution of multi-attribute range queries is mandatory to support several higher level services. Consider, for instance, the scenario described in [4] where information is collected from sensors and mobile devices with the goal of detecting traffic jams or crashes on roads. The huge amount of information gathered at the edges of the network is aggregated by considering the spatial location of the sensing nodes. Users can submit spatio-temporal multi-attribute range queries to select data related to relevant events recently occurred within a space range around them.

Even if several multi-attribute range queries supports have been proposed for peer-to-peer networks, these solutions cannot be applied directly to Fog Computing platforms since they do not meet all the distinguishing traits of this paradigm, such as the strong requirement of data aggregation, the real-time nature of several services, the high dynamicity of data collected at the edges.

Let us consider the popular solution of implementing a P2P discovery service on Distributed Hash Tables (DHTs). These approaches usually rely on data delegation, i.e. the data published by a peer is generally stored on another peer chosen according to a mapping strategy guaranteeing efficient routing. Even if this solution offers several benefits, such as logarithmic bounds for routing and the possibility of employing uniform hashing to guarantee load balancing, delegation might be undesirable in a very dynamic environment. Indeed, due to the high rate of informa-

*Email addresses:* `emanuele.carlini@isti.cnr.it` (Emanuele Carlini), `lulli@di.unipi.it` (Alessandro Lulli), `ricci@di.unipi.it` (Laura Ricci)

tion staleness the data must be updated frequently, causing large overhead to the entities handling the delegated copies [5, 6]. Furthermore, since hash functions destroy the ordering in the key value space, these structures can only support exact and one-dimensional queries in their original formulation. Numerous works extended DHTs to support the more general case of multi-dimensional range queries [2, 7, 8, 9, 10], but, in general, these solutions drops the main benefit of this approach that is the uniform distribution of data which guarantees load balancing.

Other approaches studied in the literature [11, 12, 13] directly exploit a multidimensional graph, like a Delaunay graph, with additional long range links for fast searching to support query resolution. According to these approaches, the position of a peer in the multidimensional space is defined by the attributes of the object it publishes and the object is stored on the peer itself. Even if this approach overcomes the delegation issue, the high cost of joining/leaving cannot be tolerated in a highly dynamic environment. Furthermore, aggregation strategies are not easily supported.

A further class of solutions is that of distributed tree-based overlays [14, 15, 16, 17, 10, 2] which naturally adhere to the hierarchical structure of the fog architectures. However, most of these solutions [14, 15, 17, 2] suffer the same problems of DHT-based approaches since they partition the domain of values among the peers and delegate the management of data to the node paired with the range including the data value. Furthermore, multi attributes queries are not always supported.

In this paper we present DRAGON (Distributed Range AGgregatiON), a tree-based overlay targeting the Fog Computing paradigm which integrates hierarchical data aggregation and multi-attribute range queries resolution. In DRAGON, each node publishes a set of multi-dimensional objects which have to be retrieved by queries. For the sake of simplicity, in the paper we suppose that each peer publishes a single object. Each object may represent, for instance, a remote sensing performed by a sensor or by a mobile phone. Data is collected and stored by the edge peers so that no data delegation is introduced. Peers collaboratively manage a distributed aggregation tree, where the internal nodes of the tree store a digest of the objects published at the edges. An initial data aggregation may be performed at the edge peers which in any case maintain the original data in their storage to solve queries. The aim of the aggregation is to: (i) provide an approximated view of data stored in the tree during query resolution, and (ii) limit the overhead when data is updated.

In DRAGON, the query resolution process exploites the information stored in the aggregation tree to prune the sub-trees containing few or no relevant data when resolving queries, and manages efficiently fast changing data values. The flexible architecture of DRAGON enables to plug-in different approximation functions into the nodes of the aggregation tree, so to tune the trade-offs among the complexity of the aggregation function, the frequency of data updates, and the precision in the query resolution process.

Multidimensional data is managed by exploiting *space filling curves*, which pair a single one-dimensional derived key to each data. This approach reduces both the bandwidth required for transmitting data on the overlay and the complexity of the aggregation functions.

A preliminary version of DRAGON supporting only single-attributes range queries has been presented in presented in [18]. This paper largely extends and improves from that initial result, and the main contributions can be summarized as follows:

- we propose a solution for multi-attribute range queries based on the integration of aggregation techniques and Space Filling Curves (SFCs, Section 4). We show that the choice of the Z-space filling curve allows to optimize the query resolution process by avoiding useless intersection between the query and the aggregate information stored in the digest (Section 4.3);
- we propose a failure detection mechanism to cope with network churn based on a retry mechanism (Section 4.4);
- we provide an extensive experimental analysis, by considering a realistic scenario for IoT. The experimental part includes an evaluation of the load, the ability to answer query, an evaluation of the impact of data dynamicity on digest updates as well as a comparison with MatchTree [10], a similar state of the art proposal (Section 5). From the experimental analysis we found evidence that aggregation strategies have an impact on the trade off between the number of nodes contacted during query resolution and the cost of dynamic updates of the digest. Further, we found that the optimizations helps when resolving hard queries, allowing to cut a 50% of computational overhead per node. Finally, we show that DRAGON outperforms MatchTree in term of node contacted when resolving a query.

While most approaches presented in the literature are based on computationally expensive algorithms to keep the distributed data structure consistent in presence of data updates, our proposal limits the impact of data dynamicity by design. Furthermore, to the best of our knowledge, our approach is the first one which fully integrates aggregation techniques and space filling curves to provide a flexible support for range queries.

The description of the distributed data structure and of how the distributed tree is built is given in Section 3, while we provide a detailed overview of the query-resolution process in Section 4 where we show how SFCs and aggregation techniques can be integrated in order to support multi-attribute range queries. In this section we also provide several optimizations to the basic query resolution process. An extensive experimental analysis is presented in Section 5.

| name | multi attribute | tree structure | data delegation |
|---|---|---|---|
| DRAGON | **yes** | **yes** | **no** |
| Baton [14] | yes | yes | yes |
| Q-tree [15] | yes | yes | yes |
| PHT [17] | no | yes | yes |
| P-Grid [16] | no | yes | yes |
| Saturn [19] | no | no | yes |
| MAAN [7] | yes | no | yes |
| LORM [8] | yes | no | yes |
| Squid [9] | yes | yes, implicit | yes |
| MatchTree [10] | yes | yes, on demand | no |
| IoT Discovery Service[2] | yes | yes | yes |

Table 1: Comparison of several range queries approaches

## 2. Related Work

In this section we contextualize Dragon in the area of Fog and IoT, then we present several proposals of multi-attribute-range queries developed in the area of peer-to-peer networks and a recent proposal designed specifically for IoT scenarios.

### *2.1.* DRAGON *in the Context of IoT*

Bonomi et al. [3] discuss a number of characteristics that make the Fog as a system for the management of IoT, a non-trivial extension of Grid and Cloud systems. The most important of them are location awareness, low latency, high dynamicity and the huge number of connected nodes. Furthermore, the heterogeneity of nodes which range from very simple devices with reduced computational capacity to more powerful nodes calls for a hierarchical multi-tier architecture. Even if resource discovery has been a fundamental component of grid systems, and has been a highly active research area, most of the presented proposals cannot be easily adapted to the Fog. Navimipour et al. [20] classifies the grid resource discovery systems in centralized and hierarchical system, P2P or agent based. Centralized solutions are not adequate for the Fog whose main goal is to store information at the edge of the network, close to the users. P2P-based resource discovery systems for grids generally exploit the DHT technology and are therefore based on the principle that data are not stored locally where they are generated, but storage is delegated to other nodes. Such delegation introduces extra overhead, which is especially heavy when data is highly dynamic. Typical nodes of the Fog have limited resources and finite energy and are not able to support this extra overhead. Finally, grid agent-based solutions inject in the network code fragments which are locally executed at each node, where they perform resource discovery. This solution is not suitable for Fog because some limited resources nodes may not support agent execution and because of general security issues.

Starting from these issues, we have conceived DRAGON with the goal of satisfying at least a subset of the requirements defined in [3]. The DHT in DRAGON is used only to assign identifiers to nodes, and not to delegate storage. This is a relevant point, as it allows data to always be stored at the edge of the network and to avoid the extra overhead due to delegation. Indeed, in DRAGON data modifications imply the update of the digests, but data delegation is never required. Finally, location awareness may be supported by assigning node identifiers to preserve physical proximity as it happens in [21].

### *2.2. Distributed Range Query Support*

Among the numerous approaches supporting range queries, we discuss here those more closely related to our work. In particular, we compare the proposals with respect to the following aspects: (i) support to multi attribute queries, (ii) use of a tree structure, (iii) data delegation. In particular the problem of building a distributed tree has been studied in [17][16][22][10]. A summary of the analysed approaches is shown in Table 1.

Baton [14], BAlanced Tree Overlay Network, is based on a binary balanced tree structure in which each node of the tree is maintained by a peer. Each node, both leaf and internal, is paired with a range of values and data published is stored on the peer managing the corresponding interval of values, hence data delegation is exploited. These ranges are dynamically adjusted at each node so to guarantee load balancing, with overloaded nodes transferring part of their contents to other nodes.

Q-tree [15] provides a multi-attribute based query solution for hierarchically clustered environments like tele-immersive interactive systems. Each node is assigned a range interval that specifies which items it stores and also knows the entire range of the subtree rooted at it. Data delegation is exploited to store data in the tree.

PHT [17] is similar to our work because it builds a static tree to route the query resolution, however it supports only single attribute range queries. Specifically, it builds a trie on top of a DHT by exploiting the high-level operations lookup, put and delete of the underlying DHT. The management of a key K is delegated to a leaf node whose label is a prefix of K. In order to resolve queries and perform updates, PHT requires to find the leaf of the trie, which in turn requires a variable number of DHT lookup depending on the number of unique prefixes in the trie.

P-Grid [16] builds a static trie to route the query resolution and it does not require an underlying DHT. However it resolves only single attribute range queries. In P-Grid each peer is associated with a leaf of the binary tree and for each level of the tree it maintains a reference to some other peer that does not pertain to the peer's subtree at that level. P-Grid needs a sample of data to build a balanced trie and exploits data delegation mechanism to store the data in the peers.

An instance of Saturn [19] consists of an order preserving DHT ring and a number of virtual rings where resources are distributed using a defined *Multiring Hash Function*. Range queries are solved by randomly selecting one of the virtual rings, exploring the nodes from the lower until the upper bound of the queries. As PHT, also Saturn does not provide support for multi-attribute range queries.

MAAN [7] maps objects on separate address spaces (one for each dimension), which are then commonly collapsed in a single DHT. Multi-attributes range queries are resolved considering the attribute that minimize the address space to explore, and by filtering out the results for the rest of attributes. MAAN can lead to high network overhead when values change rapidly, since it requires an update for the whole resource for each attribute. MAAN can also creates hot-spots since bulk of information can end up managed by few nodes.

LORM [8] organizes peers in a set of clusters distributed in a DHT ring, such that each cluster is responsible for the management of one attribute. Range queries are solved routing one sub-query for every attribute to the cluster responsible of the attribute and then aggregating the results. The downsides of this approach are the following: (i) nodes can be responsible of a large amount of information, causing large overhead in case of churn (due to data delegation); (ii) resources information must be refreshed periodically, increasing the network load.

Similarly to DRAGON, Squid [9] solves multi attribute range queries using a locality preserving indexing scheme based on the Hilbert SFC. The SFC index space is chosen to be the same as the node identifier space, and each peer is responsible for the data in its segment. Squid's query resolution approach can be viewed as constructing a tree and visiting it top-down, increasing the prefix by one at every level of the tree and checking if the cluster of the SFC-based index space matches the range query. However, this approach can overload the root of the trees because the peer handling the shortest prefixes of the identifier space are contacted frequently to start the query processing. In addition Squid requires a load balancing mechanism as the uniformly distribution of node identifiers leads to data unbalancing.

MatchTree [10] is a self-organizing recursive-partitioning multi-cast tree where the tree structure is built according to the query. Queries are propagated into the tree according to the goodness of the values, and results are returned aggregated and sorted by rank. MatchTree employs a set of heuristics to increase query performance and a redundant topology to support failure. Similarly to DRAGON, MatchTree provides the resolution of multi attribute range queries on top of a tree. Even if it adds interesting functionalities like ranking results, MatchTree suffers of network bandwidth consumption because a different tree must be created for every request.

Finally, [2] proposed a Discovery Service specifically designed for Internet of Things scenarios which supports multi-attribute range-queries and adopts a peer-to-peer approach for guaranteeing scalability, robustness, and maintainability of the overall system. The Discovery Service linearises multi-attributes through space filling curves and exploits an indexing PHT structure (previously presented) built on top of the Kademlia DHT overlay network.

## 3. The Aggregation Tree

DRAGON is a distributed searchable data structured organized as a binary aggregation tree. Every peer manages exactly a leaf node of the tree, which contains the data published by the peer, and may in addition manage other internal nodes of the tree.

Hereinafter we use the term *peer* to refer to a machine able to perform computation and connect to the network, whereas we use the term *node* to refer the nodes of the DRAGON tree.

### *3.1. Tree Construction*

When a new peer joins DRAGON: (i) it gets a DHT identifier, (ii) exploits the routing mechanism of the DHT for finding a peer already belonging to DRAGON, and (iii) triggers the aggregation process. The DHT address space is used to build a trie over the alphabet $\Sigma = \{0, 1\}$. Note that the DHT identifier of a peer is different from the key paired with the object it publishes and is exploited only for building the tree. Any classical DHT may be exploited to obtain a uniform distribution of the peers and, as a consequence, a balanced tree. On the other way, a locality sensitive hashing DHT may be exploited when the platform requires aggregation of spatially close data. In any case, it worth noticing that the DHT is only exploited during the bootstrap phase and not during the query resolution.

Figure 1 shows a DRAGON tree before and after the arrival of the peer $P_4$. The full join procedure goes as the following. $P_4$ first receives the DHT identifier 101 and afterwards, using the key-based routing of the DHT, it searches for a peer sharing the longest common prefix of the identifier (which is $P_1$ with id 100). Considering $P_1$, $P_4$ finds 10 as their least common ancestor (LCA). Once found the LCA, $P_4$ determines the nodes to manage. In fact, the management of node 10 is contended between $P_4$ and $P_1$. To resolve the dispute, the node is assigned to the peer managing the data item with the highest value,

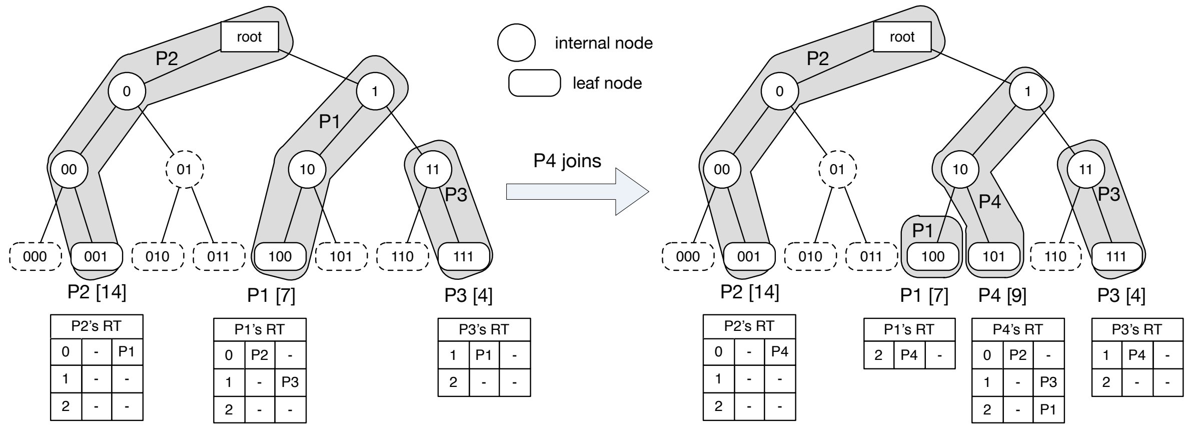


Figure 1: On the left, a DRAGON tree with 3 peers $\{P_1, P_2, P_3\}$ and $h = 3$. On the right the same tree after $P_4$ joined with identifier 101. Grey areas highlight the mapping of nodes to peers, nodes with dotted stroke are not mapped. Squared brackets contain the data value, RT stands for Routing Table.

in this case $P_4$ (the value of $P_1$ is 7 and the value of $P_4$ is 9). This strategy has been chosen according to [23], in which a similar tree is built for the resolution of exact one-dimensional queries. In addition, we performed tests that confirmed than the strategy used to resolve disputes has a marginal impact on DRAGON performances. Then $P_4$ climbs the tree until loosing a dispute at the root managed by $P_2$ (which has value 14).

In general, more complex criteria to resolve disputes can be defined to enhance certain properties. In particular, one could try to balance the number of nodes associated to peers, or reduce the number of messages exchanged by peers in particular cases (e.g. according to the distribution of the queries). These aspects, although interesting, deviate from the main scope of this work (i.e. the query resolution process) and therefore are not covered in the following.

During the join, along with nodes assignment, a peer builds its routing table, causing an update on the routing tables of the peers encountered during the climbing of the tree. Each peer maintains a row $r$ in the routing table for every level of the tree, containing two values: 1) the peer handling the node at level $r$, and 2) the sibling of the peer at level $r+1$. For example since $P_4$ manages the node with prefix 10, $P_1$ sets its parent entry at level 2 equals to $P_4$, while $P_4$ sets $P_1$ as sibling entry at level 2.

### 3.2. Aggregation Techniques

In DRAGON each peer publishes an object characterized by a set of attributes. This set of attributes is linearised by exploiting space-filling curves (as described in the following section) and the derived key is stored in a leaf of the tree. Each internal node of the tree stores a *digest* summarizing the data contained in the relative sub-tree. The purpose of the digest is to drive range queries to node that contain useful data.

Digests are updated when the derived key of a peer changes, this may happen when the value of an attribute is modified or if the number of attributes or their range of values changes. The update of digests starts from the leaf node and can potentially arrive up to the root of the tree. Note that this also applies to the join, as the entrance and update as a new value is managed in the same way. Given a tree like in Figure 1, suppose $P_1$ updates the value of its data. It sends the update to $P_4$ which first updates the digest of its assigned nodes, i.e. 10 and 1 in this order, then it forwards the update up to the tree. In the example, if the digest on 1 required an update, then the propagation would go up to the root to $P_2$. In this design, the definition of the digest represents a trade-off between the level of approximation provided (less detailed information during query resolution) and the number of updates required when data is modified. DRAGON exploits two different digest strategies, Bitvector and Q_DIGEST, briefly discussed below.

Note that if one of the two parameters characterizing the SFC, i.e. the number of attributes or the attribute value range is modified, all the derived keys are modified. In these scenarios, a full data rearrangement is unavoidable. However, the cost of this operation in our system is lower compared those systems based on the mapping of the SFC derived keys onto a DHT. In this case, an update of the parameters characterizing the SFC requires, in the worst case, a new mapping of all the data items onto the DHT nodes. Instead, in Dragon the underlying DHT is left unchanged, since the DHT is only exploited for an uniform distribution of the nodes, but no data is mapped on it. Furthermore, even if the digest stored in the internal nodes of the tree have to be modified, the structure of the Dragon tree is unchanged. The digests are updated by a bottom-up visit of the Dragon tree, starting at the leaves and modifying the digest on the path to the root. The number of digests to be updated depends on the specific digest strategy: if the digest returns a coarse approximation, the probability that the update stops at low level of the tree is higher. In this way, by choosing a proper digest strategy, it is possible to tune the trade-off between the cost of the updates and the precision of the information driving the search.

### *3.2.1. Bitvectors*

Bivectors have been originally exploited to implement routing indexes for unstructured networks [24]. A bitvector is defined by partitioning the space into $k$ intervals, and a vector $B$ is defined as $B = (b_0, b_1, \ldots, b_k)$ such that $b_i = 1$ if and only if exists a data belonging to the interval $[P_i, P_{i+1})$. Practically, in the implementation only the bits with value equal to 1 are stored.

The main advantage of bitvectors is the straightforward implementation of merging two bitvectors, which can be computed by considering their bitwise disjunction. On the other hand, the approximation introduced by bitvectors may result too coarse to effectively support the resolution of a range query. A bitvector shows if *at least* a piece of data belongs to one of the intervals, but does not provide any information about the amount of data in the intervals.

### *3.2.2. Q Digest*

The Quantile Digest (Q_DIGEST) approach [25] has been originally used as an aggregation technique in sensor networks. The main idea of Q_DIGEST is to provide an approximation of the distribution of the data by using variable size *buckets*. A single Q_DIGEST contains a set of buckets of variable size defined over a space $[1, \gamma]$. A bucket $b$ has associated an interval $[l_b, u_b]$ and a counter $c(b)$ that indicates the number of data values in the bucket. All the possible buckets of a digest are represented by a tree built on top of the space $[1, \gamma]$, in which a node corresponds to a bucket whose interval is the union of the intervals of the two siblings. Among all possible buckets, only a subset of them is kept in the digest, according to a compression parameter $c$. Given $c$, a bucket $b$ is in the Q_DIGEST if and only if these properties hold [25]: (1) $count(b) \le \lfloor n/c \rfloor$, and (2) $count(b) + count(b_p) + count(b_s) > \lfloor n/c \rfloor$; where $b_p$ and $b_s$ are respectively the parent and the sibling of $b$, and $n$ is the total number of data values.

Compared to bitvectors, which provide only the presence of data in a given interval, Q_DIGEST gives an estimation of the amount of data in an interval. Further, Q_DIGEST comes with some interesting properties, including the support for a custom level of compression, useful to balance the tradeoff between precision and size.

## 4. Multidimensional Range Query

In DRAGON each peer publishes a multidimensional object which is characterized by a set $S$ of attributes whose values define a point of the $\mathbb{R}^n$ space. A multi attribute range query is a pair $(C, K)$ where $C$ is a set of $n$ constraints $low_i \le a_i \le high_i$, one for each attribute $i \in S$, and $K$ is the number of objects satisfying the constraints required by the query.

A query originates in a leaf and climbs up the tree until enough results are found. Whenever a peer receives a query, it executes the following actions:

- checks if its local data value satisfies the range query;
- for each sibling check its digest to understand whether it contains useful data values;
- according to the amount of data values retrieved, it decides whether to terminate the query or to send it to the upper node of the tree.

This process is detailed by the *TreeSearch* algorithm, which is explained in the following section. In addition, we provide a detailed description of the *SiblingSearch* procedure, which is devoted to check whether a digest contains useful data values.

### *4.1. TreeSearch*

The TreeSearch (whose pseudo-code is presented in Algorithm 1) begins at a leaf of the tree, i.e. from the peer submitting the query, and node by node continues as a bottom-up visit of the tree.

The design choices for the TreeSearch are the following: (i) to avoid reaching the upper levels of the tree, as they can easily become bottlenecks, and (ii) to explore the parts of the tree that contain data values relevant to the query with higher probability, while ignoring those not containing useful data values. Consequently, DRAGON exploits digest information to forward queries to a promising sub-tree, and only once a sub-tree is visited, the query climbs to the upper levels of the tree.

During a TreeSearch, a peer can play the role of a *master* or *slave*. A slave explores a subtrees by forwarding the query to the siblings and waits for the results from them. Once it receives the results, it forwards the query up to the parent, until it reaches a master node. A master node can decide whether to forward the query to the upper level of the tree. Note that a master node always lies on the path from the leaf originating the query to the root, and that there is only a master node active per query at any given time.

More in detail, let us consider a generic node $P$ executing the TreeSearch. $P$ first checks if the local data value matches the query, by exploiting the LocalMatch function and in this case it updates the result (line 2 in Algorithm 1).

Afterwards, $P$ considers the siblings whose digest expose useful data values by calling the SiblingSearch (line 4). Then, until there are useful siblings and the less than $K$ items have been found, $P$ forwards the query to a subset of the siblings such that their digest exposes enough information to solve the query (lines 6-11). For example, if the query requests 3 data values, and the digests of two sibling nodes expose 4 data values each, only one of the sub-tree is contacted. This process continues recursively on every node of the sub-tree. Note that the siblings in

**Algorithm 1:** TreeSearch(Query, Mode)

```
   Input: $Query$, includes the results found so far
   Input: $Mode$, can be $master$ or $slave$
1  if $localMatch(query)$ then
2      query.updateResult(localNode)
3  end
4  C ← SiblingSearch()
5  while $C.hasMore() \wedge !query.isFinished$ do
6      C' ← removeEnough($C$, $query$)
7      forall the $sibling \in C'$ do
8          sibling.TreeSearch(query, slave)
9      end
10     waitAll()
11     query.updateResult(getAllResult())
12 end
13 if $Mode$ is $slave$ then
14     send($query$,$parent$)
15 else
16     if $query.isFinished$ then
17         send($query$,$queryNode$)
18     else
19         parent.TreeSearch(query, master)
20     end
21 end
```

this phase *are contacted in parallel*, and $P$ waits for the response and merges the updates once the visit process is completed. In the last part of the algorithm, $P$ behaves differently if master or slave. If $P$ is a slave then it returns data to its parent, i.e. the node that forwarded the query to it (line 14). Rather, if it is a master then it checks if all $K$ objects have been found and possibly sends the result to the originating peer. In case the number of found objects is less than $K$, it forwards the query to the upper level of the tree (lines 15-20).

This algorithm presents a worst case scenario when a query starting from a leaf require to contact a different master node for each level of the tree. In addition each master node contacted at level $i$ can potentially perform a child search starting at level $i+1$ until a leaf of the tree. Due to the above, if the tree height is $n$ we obtain a number of hops equal to $n*(n+1)/2$. From a qualitative analysis of the algorithm this scenario is unlike to happen for the following reasons (in the experimental analysis we run several experiments that confirmed these observations):

- child pruning: in Algorithm 1 Line 4 every node being master or slave checks the digests of its children and avoid the search on child not having relevant data for the query;
- node knowledge: every node $u$ at level $i$ in the tree has the knowledge of a node at level $i+1, i+2...$ until a leaf node (see nodes' routing table in Figure 1). For instance, during query resolution if a subtree relevant for the query is at level $i+3$ the node $u$ can skip level $i+1$ and $i+2$ and contact directly the node at level $i+3$;
- query stop: when the number of found resources is greater or equals to the number of requested resources a master node can stop the query resolution process, see Algorithm 1 Line 15.

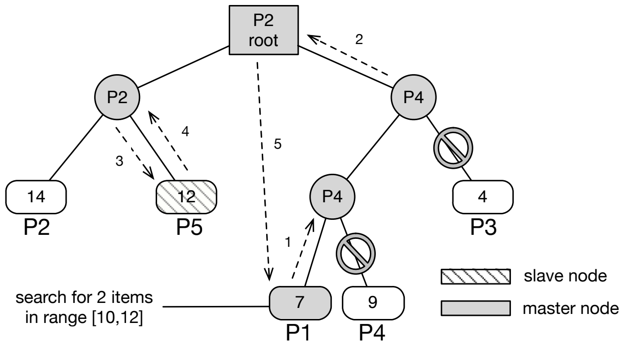


Figure 2: An example of query resolution

Figure 2 shows an example of a query resolution. $P_1$ starts the query, by executing the search algorithm as master. Since it has no siblings it forwards the query to $P_4$ (step 1 in figure), which manages $P_1$'s parent node on the tree. $P_4$ executes the search as master, by considering, its node in the upper level of the tree. Since the digest of the sub-tree rooted at $P_3$ does not expose interesting information, $P_4$ forwards the query to $P_2$ that executes the search as master node (step 2). According to the digest, $P_2$ forwards the query to $P_5$ (step 3) that executes the search as a slave node and returns the result (i.e. the data value 12) to $P_2$ (step 4). In this case, the query is not completed, but since $P_2$ is the root, the only action it can take is to return the query back to $P_1$ (step 5).

### *4.2. SiblingSearch and Space Filling Curves*

The SiblingSearch procedure is executed on the digest of a node to check whether the digest contains useful data values. In case of multi-dimensional data values, each point in the digest represents a linearisation of a value. To enable this linearisation, DRAGON exploits the *space filling approach* [26, 27] to define a bijective mapping between the $\mathbb{R}^n$ space of the objects and a linear space of *derived keys*. A space-filling curve (SFC) passes through every point of the $\mathbb{R}^n$ space once and defines a one-to-one correspondence between the coordinates of the objects in $\mathbb{R}^n$ and the one-dimensional sequence numbers of the points on the curve. Each point of the curve is called the *derived key* of the corresponding point in the multidimensional space. Since no total order may fully preserve spatial proximity in the multidimensional space, SFCs give a probabilistic guarantee that two objects located in proximity in the $\mathbb{R}^n$ space are also close on the curve. In other words, derived keys close to each other correspond to close point in the n-dimensional space. The degree of locality depends on the shape of the space filling curve. It is important to notice that exploiting a SFC allows us to apply, with the proper improvements, the digest aggregation algorithms defined for the 1-dimensional case [18] to the domain of derived keys. The linearisation of multi-dimensional points makes easy the resolution of range queries. The interval obtained by linearising the points delimiting the (hyper) rectangle

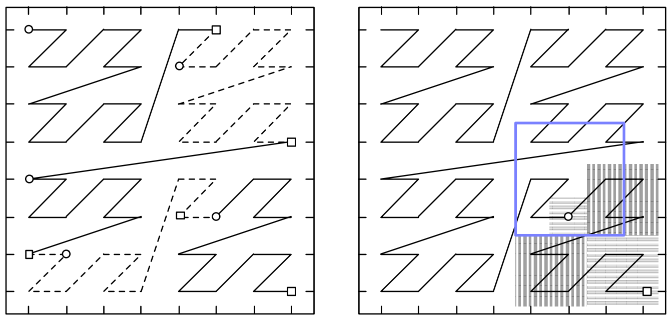


Figure 3: On the left a Z-curve divided in 5 intervals. On the right the hyper-rectangle defined by a query and the minimal Z-region quad envelope of the last interval.

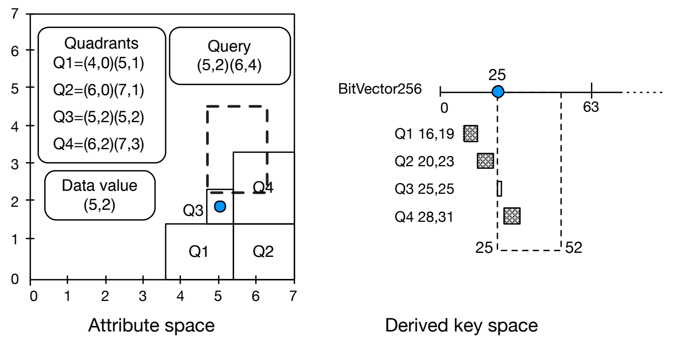


Figure 4: Data values, query and quadrants in attribute and derived key space.

of a range query corresponds to the query in the linear space. All the data values that lie in such interval match the range query.

Among several SFCs, we have chosen the Z-order SFC. The main reason is the possibility of exploiting both a cheap algorithm to compute the derived key and a straightforward method for range queries resolution. Furthermore the Z-curve exhibits a good level of locality [28]. The Z-curve visits the quadrants resulting from the recursive definition of the n-dimensional space, according to the order defined by their identifiers, at any recursion level. In this way, a bijective mapping between the quadrant identifiers and the derived keys is defined. This property is important, as it allows for the definition of relatively easy algorithms when exploiting this curve. Note that this does not hold for any SFC, for instance this is not true for the Hilbert Curve [26].

### *4.2.1. Generation of the Derived Key*

A node of DRAGON computes its derived key by exploiting a bit-interleaving algorithm [29] which takes the binary representation of the values of each attribute and interleaves bits taken cyclically from these values to construct the single derived key. For example, in Figure 4 the data value $(5, 2)$ in binary is $(0101, 0010)$. By interleaving we obtain 00011001 which corresponds to 19 in the derived key space. The resulting value is then transmitted to the internal nodes of the tree which compute the bitvector or Q_DIGEST aggregated information. Hence, each internal node stores a set of interval of derived keys values, whereas the query comes as a set of linear constraints in the n-dimensional space. An interval $[\alpha, \beta]$ of derived keys on the Z-curve, will be referred in the following as *Z-region*. Figure 3 (left part) shows 5 Z-regions, each one starting at a circled point and ending at a squared point.

### *4.2.2. Exploiting Digests for Query Routing*

The SiblingSearch is described by the Algorithm 2 and is composed by two main operations:

- the computation of the *minimal Z-region quad envelope*, which is the minimal set of quadrants of the n-dimensional space exactly covering a Z-region. These intervals are used in line 3 of the algorithm;
- the computation of the intersection between the hyper-rectangle defined by the range query and any quadrant in the *minimal Z-region quad envelope* (line 6). A rectangular query (highlighted in blue) and its intersection with the Z-region quad envelope of the last Z-region of Figure 3 (left part) are presented in Figure 3 (right part).

Figure 4 show an example of a data value, a query and a set of quadrants in the attribute and derived key space. The list of quadrants are obtained according to Skopal et al. [29], who presented a simple algorithm to compute the *minimal Z-region quad envelope*. The algorithm inserts in the *minimal Z-region quad envelope* all the quadrants whose identifiers values are between $\alpha$ and $\beta$, if they exists. The properties of the Z-curve guarantee that all the derived keys in these quadrants are between $\alpha$ and $\beta$. The border quadrants including $\alpha$ and $\beta$ are recursively divided in sub-quadrants and the same argument is recursively applied, until a quadrant including only the point corresponding to $\alpha$, respectively $\beta$ is obtained.

Each node computes the minimal Z-region quad envelope for each interval of derived keys in its digest. This phase is executed once, when a node joins the system, and must be re-executed each time the information included in the digest is modified due to peers joining and leaving the tree. Note that while this information is computed once and is independent from a specific query, the intersection between the query and the *minimal Z-region quad envelope* must be executed each time a query is received by a peer to check the existence of a match between the information stored at the digest and the range query.

The aim of the SiblingSearch operation is then to check if exists a match between a quadrant belonging to the minimal Z-region quad envelope of the Z-region $[\alpha_{digest}, \beta_{digest}]$ included in the digest of a sibling and the hyper-rectangle of the range-query. In the example of Figure 4, among the four quadrants, only Q3 and Q4 intersect the range-query. Note that this does not guarantee that the subtree rooted

**Algorithm 2:** SiblingSearch(Query)

```
Input : $Query$, includes the results found so far
Input : $sibling.envelope$, contains quadrants of minimal
        Z-region envelope
Output: $siblingsMatching$, a list of sibling matching $Query$
1  forall the $sibling \in routingTable.siblings$ do
2      intervalValid ← false
3      while $sibling.envelope.hasMoreInterval() \wedge \neg\ intervalValid$
       do
4          interval ← sibling.envelope.nextInterval()
5          if $interval.dataEstimation > 0 \wedge$
           $preliminaryCheck(interval)$ then
6              hquadIntersection ← intersect(Query,
               interval.zRegionEnvelope)
7              if $hquadIntersection = true$ then
8                  intervalValid ← true
9                  siblingsMatching.add(sibling)
10             end
11         end
12     end
13 end
14 return siblingsMatching
```

at the sibling will include a query match, because the digest only approximates the real distribution of data in the sub-tree.

### 4.3. SiblingSearch Optimizations

We defined several optimizations for the SiblingSearch described in the previous section. The first optimization exploits the basic property of the Z-curve described in Section 4.2. Consider the sequence $S$ of derived keys belonging to the hyper-rectangle defined by the range query. The minimal derived key in $S$, $\alpha_{query}$, corresponds to the vertex of the hyper-rectangle paired with the smaller value of each constraint, the same argument is applied to find the largest derived key value, $\beta_{query}$. This property does not hold for other Space Filling Curves, like the Hilbert one. By exploiting this property, we define the *Outside* optimization which may avoid the computation of the intersections between the query and the minimal Z-region quad envelope. Let us consider an interval $I = [\alpha_{digest}, \beta_{digest}]$ in the digest of a node. If $\alpha_{query} > \beta_{digest} \vee \beta_{query} < \alpha_{digest}$, no value in $I$ satisfies the range query. Algorithm 2 shows the function *SiblingSearch* where the optimization previously described is defined by the function *preliminarycheck*.

Another optimization, called *Sibling*, evaluates if the same interval of derived keys is present in the digests of more than one sibling, in this case the intersections between the query and the quadrants of the minimal Z-region envelope characterizing this interval is executed only once.

### 4.4. Coping with churn

In DRAGON nodes can detect and react to churn during query resolution. Failures of master and slave nodes are detected by different mechanisms. As shown in Algorithm 1, a master node $m$ is always contacted from another master node or from the query node, hereafter called $pred(m)$, i.e. the predecessor of $m$. We introduce the two following messages to let $pred(m)$ detect the failure of $m$:

- when $m$ forwards the query to the next master node it also notifies $pred(m)$;
- when $m$ decides to terminate a query it contacts both the query node and $pred(m)$.

Given the above $pred(m)$ can detect a failure of $m$ if it is not receiving one of the two messages in a configurable amount of time. The node detecting a failure must contact the query node to communicate the premature end of the query.

The detection of the failure of a slave node does not introduce additional messages. A node, either master or slave, can detect the failure if the slave is not responding within an amount of time defined by a predefined threshold. The node detecting the failure of a slave node just skips the node and continues the visit with further nodes.

In order to improve the number of resources found in case of temporary tree inconsistencies due to churn, a retry mechanism is executed when a peer failure is detected. This mechanism is based on the two following considerations:

- since the DRAGON tree is built on top of a DHT, the lookup mechanism of the underlying DHT may be exploited to find a peer at random to restart a query;
- each peer belonging to the underlying DHT is paired with a node of the aggregation tree.

When the query node (i.e. the node that starts the query) receives a notification of the end of query or a failure is detected but not all the $K$ requested resources have been found, the query node can choose to continue the search process by selecting at random an identifier $ID_{rand}$ in the DHT address space and by using the lookup mechanism of the DHT to find the peer $P_{retry}$ that is successor of $ID_{rand}$, where the query is restarted. We introduce a retry parameter $R$ defining the maximum number of times the retry mechanism can be executed. The retry mechanism is activated only if less than $R$ retries have been executed and all the $K$ requested resources have not been found yet.

## 5. Experimental Evaluation

This section presents a selection of the most relevant experimental results for DRAGON. The main goals of the experiments were the following:

- compare the digest strategies;
- evaluate the optimizations introduced in Section 4.3;
- evaluate the impact of a high number of dimensions (up to 7);
- evaluate the load during query resolution and the impact of data updates on the amount of digest information to be modified;

- evaluate the performance in presence of churn;
- compare DRAGON with MatchTree [10].

All the results were obtained through simulations. We developed a prototype of DRAGON using the *Overlay Weaver Toolkit, OW* [30], which provides a common high level *API* to develop distributed services based on DHTs. All the simulations were conducted by exploiting the network emulation of OW, in which every node runs in an independent thread.

## *5.1. Experimental Environment*

To emulate an IoT-like scenario, we built a three dimension dataset considering the positions of the postboxes in the United Kingdom. In this context, we can image the postboxes as nodes connected to the Internet, having as attributes their position and the amount of mails contained. By organizing the mail retrieval from postboxes with DRAGON, a post office could perform geographical queries to optimize the retrieval, like: "gives me K full mail boxes in this area".

We scraped the latitude and longitude of around 57 thousands Royal Mail postboxes from [31] and normalized their latitude and longitude in the range $[0 : 2^{14} - 1]$, so to obtain enough precision in the location representation. We then added a third dimension, the *weight*, representing the amount of mails contained in a postbox. The weight follows a Normal distribution with $\mu = 2^{14}/4$ and $\sigma^2 = 2^{14}/8$ (we cut the values below 0 and above $2^{14}$). Figure 5 shows the distribution of normalized locations and weights. We also generated four synthetic datasets with 10000 items having respectively 4, 5, 6 and 7 dimensions . All the items are in the range $[0 : 2^{14} - 1]$ for each dimension.

In order to measure the performance under different kinds of queries, we characterize multi-attribute range queries by two parameters, the difficulty $d$ and the size $K$. We defined the difficulty $d \in \{3, 6, 12, 24, 48\}$ of a query as the percentage of items in the system matching the query. The

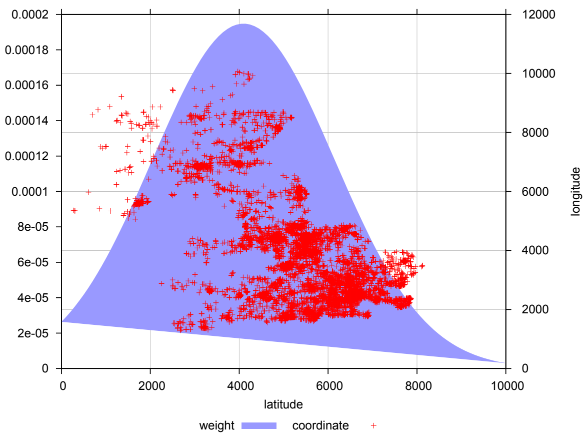


Figure 5: Distribution of latitude and longitude and frequency of weight distribution

size $K \in \{3, 6, 12, 24\}$ indicates the minimum resources to be found (if available). For example, in a network with $N = 10000$, a query with $d = 3$ and $K = 24$ requires to find at least 24 of the 300 items that match the query.

## *5.2. Comparison of Aggregation Functions*

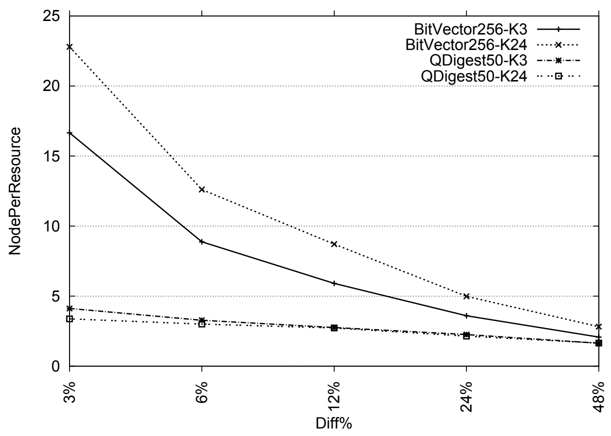


Figure 7: Aggregation comparison: NodePerResource with different value of K

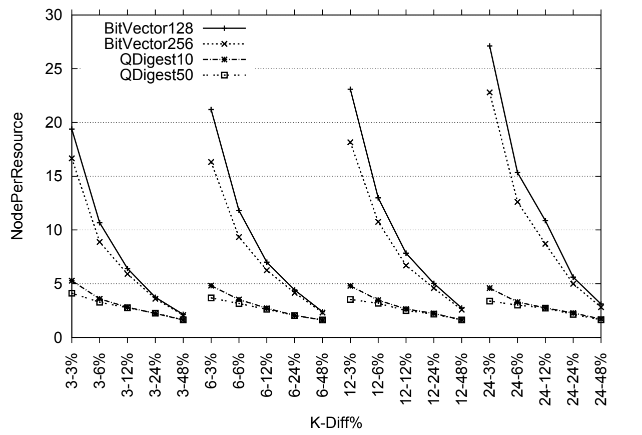
This set of experiments compare Q_DIGEST and BitVector considering the postboxes dataset. We recall that the digest plays an important role in the discovery process, as it gives to DRAGON the ability of pruning useless branches of the tree.

We randomly selected $N = \{2500, 5000, 10000\}$ postboxes from the postboxes dataset. Since the size of the attributes domain of the dataset, and according to Section 4.2, we created a SFC with a domain size $\iota$ of $2^{14\times3} = 2^{42}$. In our test we use a DHT address space of 160 bits resulting in $2^{160}$ possible different identifiers, having a tree structure with a maximum of 160 levels (note that the DHT space is not related with the domain size of SFC; the former is used to assign unique ID to nodes, the latter represents the internal value of a node). However, given the uniformity of the SHA1 assignment and that $N \ll 2^{160}$, in practice all the nodes are compacted in the highest 20 levels of the tree.

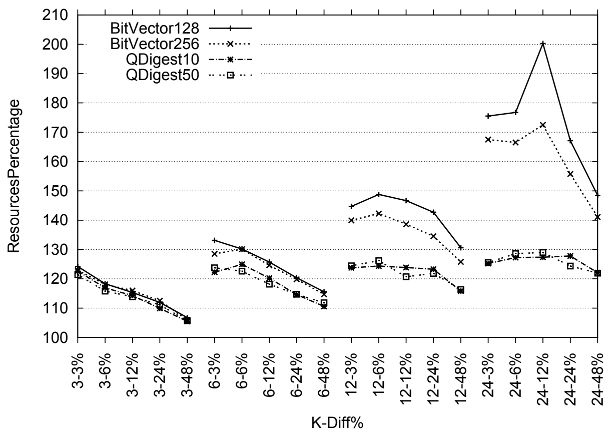
We then compared four different types of digest strategies:

- Q-10: Q_DIGEST configured with $c = 10$ (where $c$ is the compression parameter);
- Q-50: Q_DIGEST configured with $c = 50$;
- BV-128: BitVector dividing $\iota$ set in 128 equal size partitions;
- BV-256: BitVector dividing $\iota$ set in 256 equal size partitions.

In order to measure the performance, we considered the following two metrics:



(a) Aggregation comparison: NodePerResource with 10000 nodes



(b) Aggregation comparison: Resources percentage

Figure 6: Comparison of the digest strategies

| $K = 12$ | Plain | | Outside | | Outside + Sibling | |
|---|---|---|---|---|---|---|
| d | Q-50 | BV-256 | Q-50 | BV-256 | Q-50 | BV-256 |
| 3 | 1288.54 | 1243.66 | 852.26 | 849.03 | 827.79 | 754.15 |
| 6 | 730.51 | 661.08 | 460.4 | 428.75 | 448.76 | 355.89 |
| 12 | 393.49 | 336.72 | 283.89 | 245.33 | 276.02 | 196.68 |
| 24 | 216.99 | 168.2 | 196.76 | 151.05 | 192.85 | 115.54 |
| 48 | 114.63 | 85.34 | 112.2 | 79.23 | 110.76 | 72.03 |

Table 2: Average number of intervals considered during query resolution

- **NodePerResource**: quantifies the number of nodes that must be contacted to retrieve one resource. It is computed as the number of nodes contacted divided by $K$, with lower values corresponding to a better result (i.e. less nodes contacted to solve the same query).
- **ResourcesPercentage**: measures the percentage of resources found of the $K$ requested. In a sense, this metrics measures the accuracy, and with values under 100%, DRAGON was unable to answer the query successfully.

We conducted the test by running every combination of $d$ and $K$ from 100 queries, originated at random nodes. All the results are the average of 30 independent runs (in total, 3000 queries for each $\{d, K\}$ pair).

In Figure 6a and 6b the X axis corresponds to the $\{d, K\}$ pairs, while the Y axis is the average NodePerResource, respectively, ResourcesPercentage. Figure 6a reports the result for a network of 10000 nodes and shows the relevant impact of the aggregation function on the number of visited nodes. Q_DIGEST outperforms the BitVector, because the BitVector has a coarser grain precision resulting in more false positives. Q_DIGEST exploits always less than 5 nodes to find one resource, and it approaches 2 for easier queries. This is evident in Figure 7 in which Q_DIGEST and BitVector are compared with respect to selected values of $K \in \{3, 24\}$ and of $d$. Results suggest that NodePerResource maintains the same ratio when increasing $K$ for Q_DIGEST. Instead BitVector decreases performance when the number of requested resources $K$ increases.

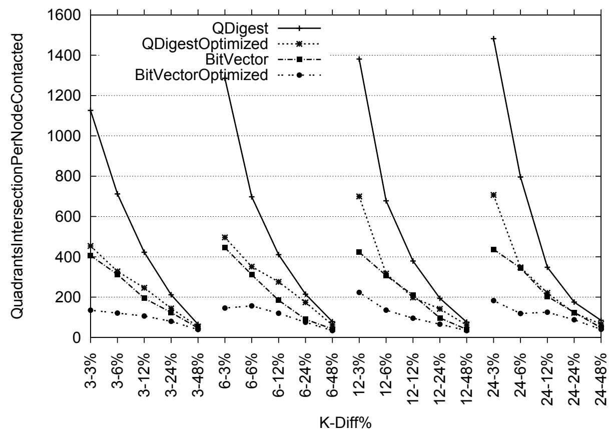
In terms of ResourcesPercentage, all digests find a number greater than or equal to the requested resources in all the configurations. However, from Figure 6b, is clear that BitVector finds more resources than Q_DIGEST. This confirms that BitVector tends to underestimate the presence of useful data items in a sub-tree, as suggested by the NodePerResource measurements. Also, BitVector floods more than necessary during sub-tree traversals, with an increment of the flooding with larger K.

## 5.3. Evaluation of optimizations

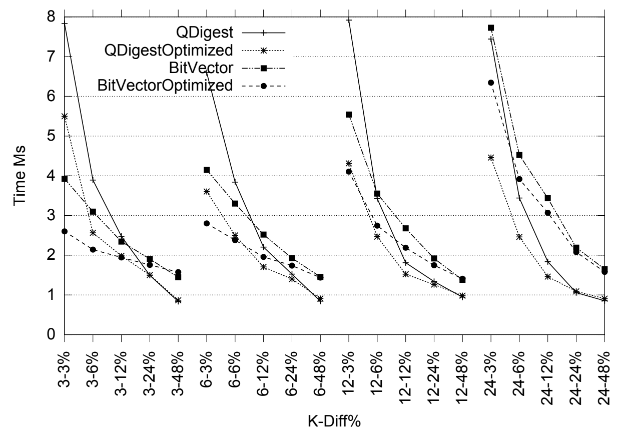
In this section we evaluate how the *Outside* and *Sibling* optimizations introduced in Section 4.3 help to reduce the computational effort on nodes in the query resolution process.

Table 2 shows the number of intervals (where more intervals means more computation needed) on average to solve a query. The results were obtained considering BV-256 and Q-50 and $K = 12$. From the results we deduct the following considerations:

- the Outside optimization has a positive impact overall, in particular for difficult queries in which DRAGON

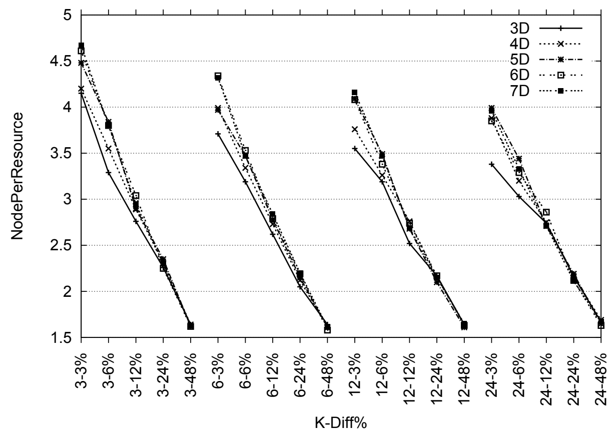


(a) Quadrants intersections per node

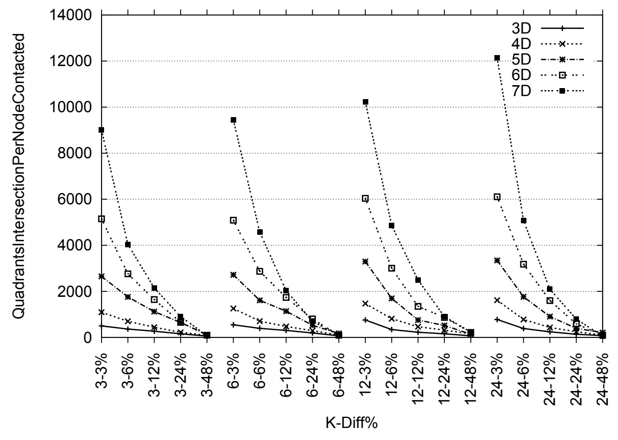


(b) Quadrants intersections time

Figure 8: Evaluation of optimizations



(a) NodePerResource metric comparison



(b) Quadrants intersection per node

Figure 9: Comparison of different multidimensionality

evaluates 30% less intervals for both Q_DIGEST and BitVector;

- the Sibling optimization mostly improves BitVector. This result was expected, because using BitVector the derived key domain is divided for every node in the same manner. Using Q_DIGEST, and having a finer grain representation, we have a smaller possibility to find a sibling having the digest sharing intervals with another sibling;

- BitVector considers always fewer intervals with respect to Q_DIGEST. This is of particular importance because Q_DIGEST contacts a smaller amount of nodes than BitVector. Due to this, when using Q_DIGEST every node must analyse a number of interval considerably greater than with BitVector.

The last consideration is highlighted in Figure 8a. It presents the average number of intersections between the queries and the quadrants performed by every node involved in the resolution process. We compared BitVector and Q_DIGEST using no optimization and considering both the Outside and the Sibling optimization. Every intersection has the cost of two integer summations and two integer comparison. From the figure is evident that the number of intersections is significantly greater using Q_DIGEST. In addition is remarkable how the two optimizations together can significantly reduce the number of intersections. For example using BitVector, with $K = 12$ and $d = 3$ and without optimizations we counted more than 400 intersections per node, instead introducing the two optimizations we counted around 200 intersections per node. Q_DIGEST for the same configuration requires near 1400 intersections without optimizations and 700 with optimizations. Also note that Q_DIGEST optimized performs similarly to BitVector not optimized in terms of intersections per node.

In Figure 8b we evaluated the average time in millisecond spent by each node to generate quadrants and executing the intersections. The generation code was executed sequentially by using a single core of a multi-core machine. The results show a similar trend to the number of quadrants intersection (see Figure ??), in which to an increase of $K$ corresponds a decrement of the time spent for the generation and intersection. It worth noticing that both Q_DIGEST and BitVector gain advantage from the optimizations, and that in any configuration we obtained an average computational time less than 8 milliseconds.

### 5.4. *Evaluation of multidimensionality*

In this section we evaluate DRAGON with datasets of different dimensions, from 3 to 7. How DRAGON reacts to high dimensions is relevant, since it is well known that linearisation techniques may be problematic when managing more than few dimensions [32]. In the context of DRAGON, more dimensions can lead to increased computational effort per node and more nodes contacted during query resolutions. In this section we explore both these aspects. All the experiments presented in this section are performed using the Q-50 digest strategy. We choose this configuration as the *worst case*, because, as seen in Section 5.3, it is the one with the higher computational cost.

Figure 9a shows that to an increment of the number of dimensions corresponds an increment of the number of nodes contacted to find the same amount of resources. However, despite this increment, the load on the systems remains acceptable. Indeed, Q_DIGEST preserves a good value for the NodePerResource metric also with 7 dimensions. For example, in the (worst) case of a query with $K = 3, d = 3$ in the dataset with 3 dimensions, 4.2 nodes on average are required to find a resource, while this values grows to 4.6 for the dataset with 7 dimensions.

More interesting results come from the analysis of quadrant intersections per node (shown in Figure 9b). Recall that the tests were running with all the optimizations enabled. As expected, the number of quadrant intersections to solve a query grows at higher dimensions. This can be explained by considering the differences in the domain size. With 3 dimensions the domain size is in the order of $2^{3\times14} = 2^{42}$, whereas with 7 dimensions the domain size is $2^{7\times14} = 2^{98}$. With a bigger domain size the number of quadrants to resolve a query increases. Therefore, there is an increment of the number of intersections needed to verify if a digest interval matches a query.

Figure 10 depicts the storage demands to handle 1000 and 10000 quadrants. On the X-axis is presented the number of dimensions, and on the Y-axis the storage space in KBytes required to store the quadrants. We evaluated the space required to store the quadrants instrumenting the JVM [1], and we obtained an implementation-specific approximation of the amount of storage consumed by the specified objects. Recall that to store a quadrant we need to save two integers for each dimension, each integer representing the lower and upper bound for that dimension.

[1]`http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`

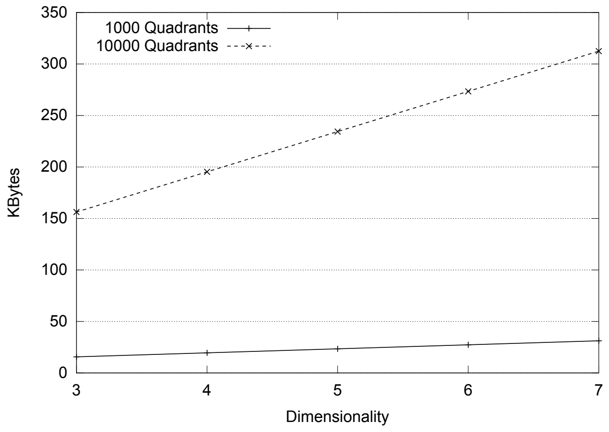


Figure 10: Storage demands per node

As expected, the storage demands increase when increasing the quadrants dimension. Specifically to store 10000 3D quadrants are required 150 KBytes, and around 300 KBytes to store 7D quadrants. Also consider that 10000 quadrants is a worst case scenario, in fact with 3 dimensions we obtained on average less than 1000 quadrants intersections using Q_DIGEST optimized, which resulted in around 10 KBytes of storage requirements.

### 5.5. *Evaluation of load*

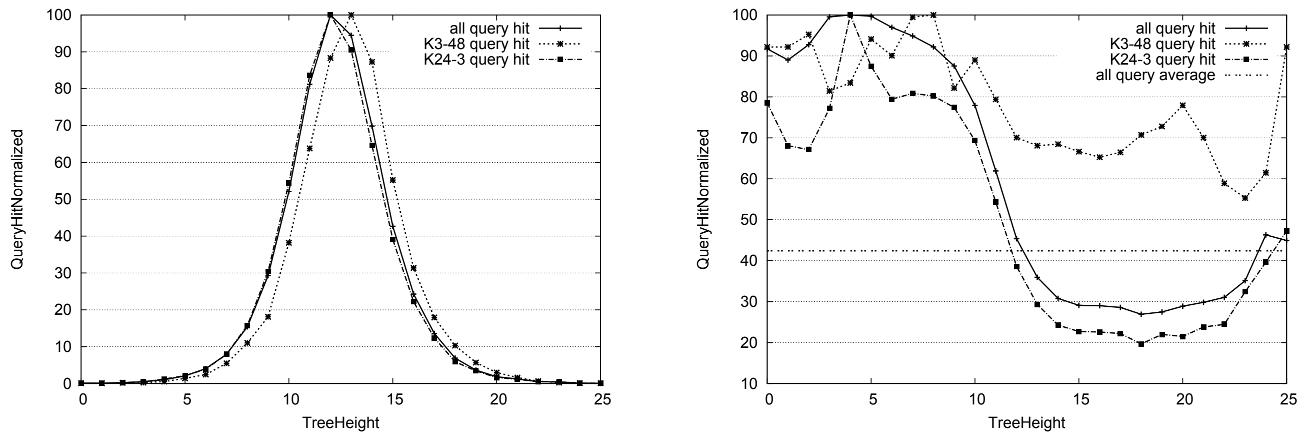
When distributed discovery services are organized as a tree, it is important to evaluate the load imposed on the nodes. In our context the load has been measured considering the number of queries elaborated by a node. Here we compared the outcomes of the easiest and hardest query, respectively $K = 3$ $d = 48$ and $K = 24$ $d = 3$. We generate 3000 different queries for every combination of $d$ and $K$.

Figure 11a shows the load for each level of the DRAGON tree. The X-axis corresponds to the tree height, with the root at $x = 0$. The Y-axis corresponds to the load. All the values are expressed in percentage according to the maximum load measured.

The levels between 0 and 10 are the higher in tree (closer to the root) and the figure shows that a small amount of queries reaches those levels. From the figure is also clear how the "middle" levels are the most loaded part of the tree in the query resolution process. Note also the lower load corresponding to levels greater than 15. This is due to the fact that those levels have lesser nodes with respect to levels ranging from 10 to 15 because all the nodes are compacted in the highest 20 levels of the tree due to the uniformity of the SHA1 assignment.

Also, the figure suggests that the easiest query climbs the tree less than the hardest one. This can be seen as the easiest query imposes more load on the lowest level of the tree.

Figure 11b shows the same outcomes but considering the average queries received per node at the same level of the tree. It is interesting to notice that, even if the total



(a) Load per level (b) Load per node

Figure 11: Evaluation of load

load on the middle levels of the tree is higher then the other levels (as seen in Figure 11a), the load per node in the middle levels is lower. This is due to the fact that the middle levels present the higher number of peers (the peers are compacted in the higher 20 levels of the tree) and the load is distributed between all the nodes of the middle levels.

As said above, it is important to evaluate the node on the highest levels of the tree. We can see that the root level has a small fraction of the load with respect to the middle levels of the tree, but instead the root node has a greater load. However, the impact is acceptable not only for the root but in general for the higher levels of the tree. For example, with the hardest query the load on the higher levels is 4x the load on the lower levels, and just 2x the average load of the system. The differences between levels flatten for the easiest query, where the difference between the most and the least loaded level is 1.5x.

## *5.6. Churn tolerance*

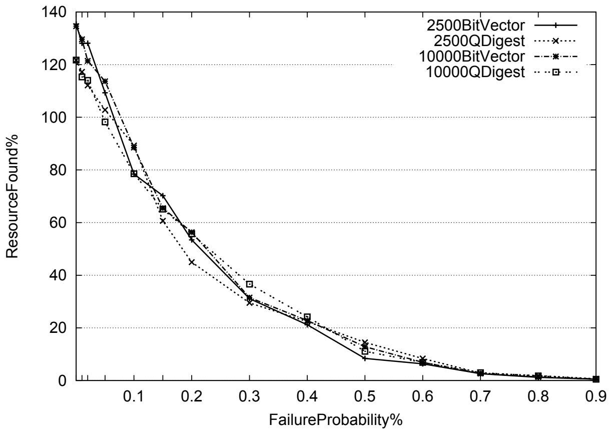
To test the efficiency of the fault tolerance mechanism we performed experiments in scenarios involving churn. Churn was modelled statically, by removing a given percentage of randomly selected nodes from the network before running a batch of queries. We consider this kind of churn because, as seen in Section 5.2, DRAGON can always find the required resources when all nodes are fully functional in the network. This scenario shows how DRAGON can route the query when parts of the tree structure is not available.

The tests were executed with a query $k = 12, d = 12$ as we think this is a good representative for the scenarios tested before. The metrics is ResourcePercentage and each point was generated analyzing 30 different queries, each query starting from 100 different nodes resulting in a total of 3000 queries per point. The outcomes of the experiments, which illustrate various settings for the retry parameter and churn rates, are shown in Figure 12a and 12b. Figure 12a shows that in presence of churn the ResourcePercentage is not affected by the aggregation strategy used. Indeed, Q_DIGEST and BitVector have a similar behaviour, because, even if information is aggregated differently, both strategies share an analogous tree structure. In Figure 12b is clear how employing a strategy based on retries (see Section 4.4) amortizes the failure on the tree structure. For example, with a 20% failures, a 1-retry strategy guarantees a 20% more resources, whereas a 2-retry strategy yields an additional 20%.

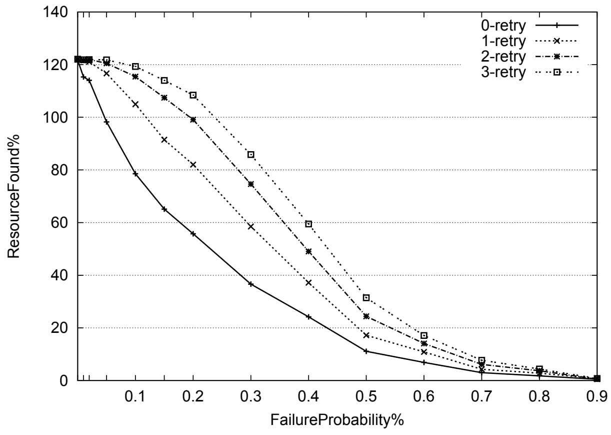
In order to evaluate the cost associated with the retry strategy in case of churn, Figure 13 shows the number of nodes contacted during the retries with different retry strategies. We observed that the number of nodes contacted when using a retry mechanism increases in case of node failure but only marginally with respect to a no churn case. The maximum number of node contacted using 3-retry with a 10% failure probability is 33, whereas in the no churn case with 0-retry we obtained a value of 30. Note that (see Figure 12b) with a 3-retry strategy, DRAGON keeps the number of found resources above the 100% up to a 20% failure probability. Hence, we can conclude that it is acceptable to have a system that complete query requests even in high churn case at the expenses of a marginal increment of nodes contacted in case of failures.

## *5.7. Evaluation of data dynamicity*

The purpose of this experiment is to analyse the impact of dynamicity due to key updates in DRAGON. The update of a key generally requires the propagation of the new digest information up to the tree. In turn, it triggers the update of the digests paired with a subset of the DRAGON nodes (as explained in Section 3.2). This experiment analyses the behaviour of DRAGON under frequent updates. Each node in the network submits an update, for a total of respectively 2500 and 10000 updates in a network with 2500 and 10000 nodes. We counted the number

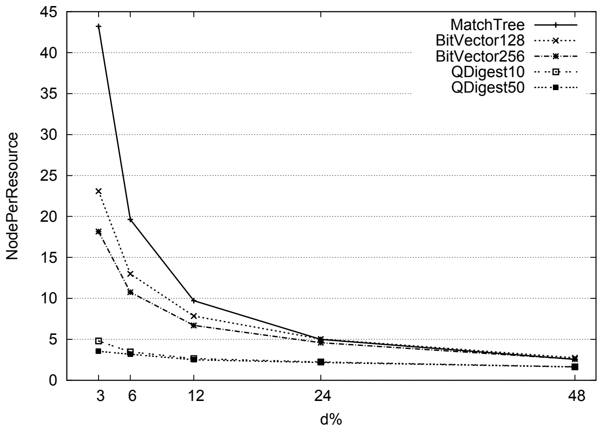


(a) Churn: comparison of aggregations and different network size

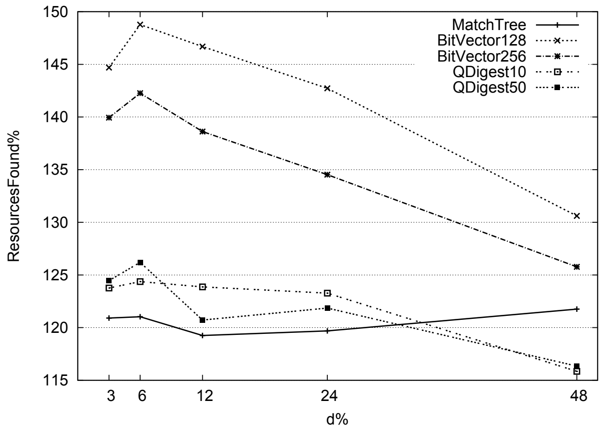


(b) Churn: retry evaluation

Figure 12: Evaluation of churn during query resolution and node contacted during change action.



(a) MatchTree vs DRAGON: NodePerResource



(b) MatchTree vs DRAGON: Resources percentage

Figure 15: MatchTree [10] vs DRAGON

of nodes involved into the stabilization process. Results are presented in Figure 14 by showing the CDF of the nodes involved in the updates. It can be seen that the 80% of updates involve less than 5 nodes with BitVector and less than 10 nodes with Q_DIGEST.

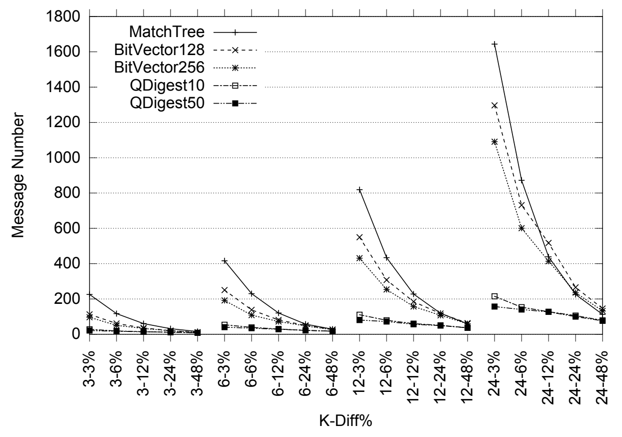
It is worth mentioning the trade off between the degree of approximation introduced by the digests and the behaviour of the update operation. The BitVector contacts on average less than 6 nodes in the 90% of the updates, instead with the Q_DIGEST the percentage drops down to 70%. Also, the number of updates required by Q_DIGEST is inevitably larger than those required by BitVector because its accuracy is higher. As a conclusion, we can confirm that even if the performance of Q_DIGEST in terms of contacted nodes is worse than that of BitVector, Q_DIGEST still represents the best compromise when considering the performance of the search versus the update operations. In addition, for the Q_DIGEST the number of contacted nodes scales when increasing the network size. Indeed, Q_DIGEST keeps the 90% of updates below 13 nodes contacted for a network of 2500 nodes and below 15 for a network of 10000 nodes.

### *5.8. Comparison with MatchTree*

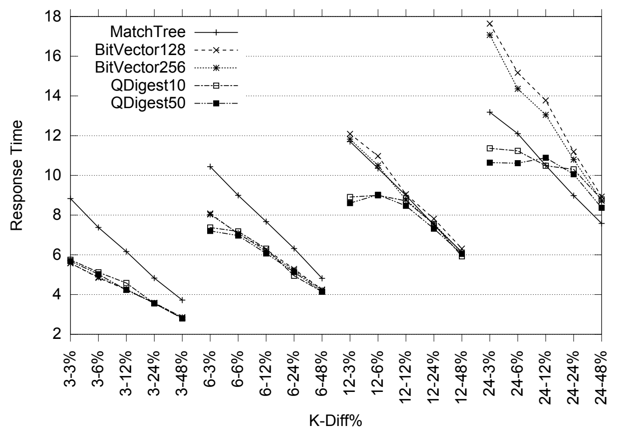
We performed a comparison with MatchTree proposed by Lee et al [10], using the code provided by the authors on GitHub[2]. We choose MatchTree because, similarly to us, it supports multi attribute range queries requesting a fixed number of resources. In addition MatchTree builds a tree on demand over a DHT, therefore sharing with DRAGON a similar structure. We select the MatchTree Sub-Region mode to solve the query because, among the methodologies presented, it is the one that minimizes the network overhead.

The experiments compared DRAGON and MatchTree considering the two metrics presented in Section 5.2, i.e.

[2]`https://github.com/kyungyonglee/social_overlay_network_simulator`



(a) MatchTree vs DRAGON: Message Number



(b) MatchTree vs DRAGON: Response Time

Figure 16: MatchTree [10] vs DRAGON

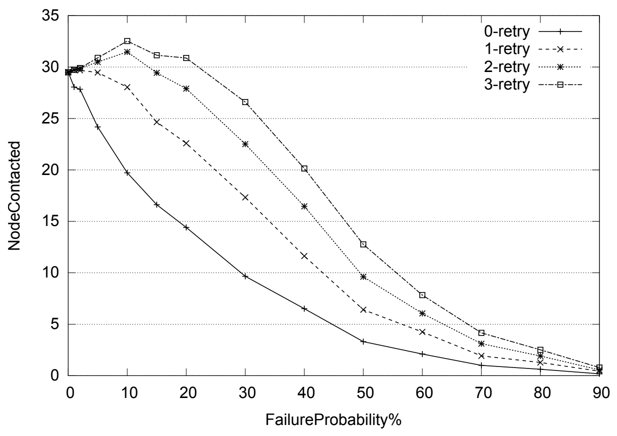


Figure 13: Churn: node contacted during retry

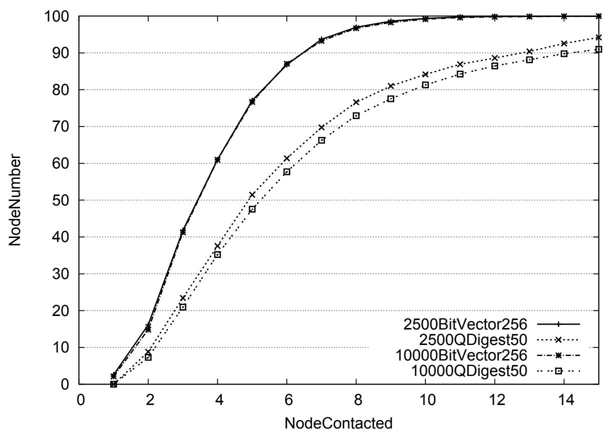


Figure 14: Change key: comparison of aggregations and different network size

NodePerResource and ResourcePercentage. Results are presented in Figure 15a and 15b. The test is performed with $K = 12$ because as seen in previous section these two metrics are not affected by the variation of $K$. The network size is fixed to $n = 10000$. MatchTree is instantiated with the postboxes dataset presented in Section 5.1. For each of the values of $d$, we run 3000 queries, each query starting from a random node.

Figure 15a presents results for the NodePerResource metric. The result shows how DRAGON contacts fewer nodes than MatchTree to solve queries. This is more evident when the query difficulties is higher. For example for queries satisfied by the 6% of the nodes, MatchTree requires to contact 20 nodes to find a useful resource, DRAGON with BitVector digest 10-15 and DRAGON with Q_DIGEST less than 5. Figure 15b shows the ResourcePercentage metric. DRAGON using BitVector is the solution with the higher average of resources found. In the other cases, MatchTree and DRAGON using Q_DIGEST have similar result.

Figure 16a present the comparison related to the message number, intended as the overall number of messages transmitted in the system to solve a query. We observed that DRAGON performs better than MatchTree, especially in case of hard queries. It also interesting to note how the two digest strategies perform when increasing $K$. BitVector is linear with $K$ (i.e. with double $K$ also the number of message used by BitVector doubles), while Q_DIGEST yields still a few number of messages even with an increasing $K$.

Figure 16b shows the comparison in terms of the response time. The response time is measured as the average number of hops visited sequentially during the query resolution process. For DRAGON we sum the number of master nodes contacted and the longest chain of child nodes contacted from each master node; for MatchTree we consider the tree height of the dynamically generated tree. We observed that Q_DIGEST has a lower response time for the most difficult queries, while MatchTree and Q_DIGEST are close for large $K$ and $d$. Similarly, BitVector has a lower response time when $K = 3$ or $K = 6$ and in general higher for

$K = 24$. This can be explained with the fact that BitVector is less precise, so incurring in less child pruning and more approximated information in the routing tables. In order to evaluate if the worst case explained in 4.1 happens frequently, we also computed the 95% confidence interval of the mean values for $K = 24$ and results are presented in Table 3. The results show some interesting insights: (i) in all the cases the confidence interval does not increase for easier queries; (ii) MatchTree is the system having less variability around the mean; (iii) Q_DIGEST has not high confidence values, suggesting that although having larger variability with respect to MatchTree the worst case scenario is limited and the values are not fluctuating a lot around the mean values.

As a conclusion, MatchTree demonstrates to be as precise as DRAGON, regarding the number of resources returned, when DRAGON uses Q_DIGEST. However, the traffic generated by MatchTree is the 500% of the traffic generated by DRAGON using Q_DIGEST for query having $d = 6$ and around the 250% for $d = 12$.

| d | 3 | 6 | 12 | 24 | 48 |
|---|---|---|---|---|---|
| Q_DIGEST | 0.52 | 0.51 | 0.54 | 0.46 | 0.38 |
| BitVector | 0.79 | 0.67 | 0.58 | 0.51 | 0.4 |
| MatchTree | 0.15 | 0.17 | 0.16 | 0.14 | 1.16 |

Table 3: Response Time: 95% Confidence Interval for K = 24

## 6. Conclusion

The issue of distributed range queries has been widely studied in the past decade from the P2P community. However, recent systems, like IoT, still demand for an effective range query support. These fields put the accents on a fast and light query resolution, in which each node publishes its own data without delegation. DRAGON is an effective solution to support distributed range query processing in such fields. It combines SFCs and an aggregation tree to build a support for multi dimensional range queries. The query resolution algorithm prunes the part of the tree that contains few or no relevant information, resulting both in a faster response and in contacting a limited amount of nodes. Experimental results have shown the effectiveness of DRAGON in terms of churn resilience, ability to resolve hard queries due to the optimizations and a lower number of nodes contacted when compared with a similar approach in the state of the art.

We plan to further refine DRAGON by experiment with other aggregation strategies, like Minimum Bounding Rectangle, Bloom filters or Wavelet. In addition, more complex query strategies can be studied, including the ability to define the trade-off between computational load and precision during query resolution, and the decomposition of the query in subqueries to allow a parallel resolution of the queries. Finally, we plan to test DRAGON in a distributed environment like PlanetLab, to further validate our results in a realistic setting.

## References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer networks 54 (15) (2010) 2787–2805.

[2] F. Paganelli, D. Parlanti, A dht-based discovery service for the internet of things, Journal of Computer Networks and Communications 2012.

[3] B. Flavio, A. M. Rodolfo, N. Preethi, J. Zhu, Fog computing: A platform for internet of things and analytics, in: Big Data and Internet of Things: A Roadmap for Smart Environments, 2014, pp. 169–186.

[4] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe, Mobile fog: A programming model for large-scale applications on the internet of things, in: Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing, MCC '13, ACM, New York, NY, USA, 2013, pp. 15–20.

[5] E. Carlini, M. Coppola, L. Ricci, Probabilistic Dropping in Push and Pull Dissemination over Distributed Hash Tables, in: Proc. of the 11th Int. Conf. on Computer and Information Technology (CIT), IEEE, 2011, pp. 47–52.

[6] E. Carlini, M. Coppola, D. Laforenza, L. Ricci, Reducing traffic in dht-based discovery protocols for dynamic resources, in: Grids, P2P and Services Computing, Springer, 2010, pp. 73–87.

[7] M. Cai, M. Frank, J. Chen, P. Szekely, Maan: A multi-attribute addressable network for grid information services, Journal of Grid Computing 2 (1) (2004) 3–14.

[8] H. Shen, C.-Z. Xu, Leveraging a compound graph-based dht for multi-attribute range queries with performance analysis, IEEE Transactions on Computers 61 (4) (2012) 433–447.

[9] C. Schmidt, M. Parashar, Squid: Enabling search in DHT-based systems, Journal of Parallel and Distributed Computing 68 (7) (2008) 962–975.

[10] K. Lee, T. Choi, P. O. Boykin, R. J. Figueiredo, Matchtree: Flexible, scalable, and fault-tolerant wide-area resource discovery with distributed matchmaking and aggregation, Future Generation Computer Systems 29 (6) (2013) 1596–1610.

[11] O. Beaumont, A. Kermarrec, L. Marchal, E. Riviere, Voronet: A scalable object network based on voronoi tessellations, in: 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA, 2007, pp. 1–10.

[12] R. Baraglia, P. Dazzi, B. Guidi, L. Ricci, Godel: Delaunay overlays in P2P networks via gossip, in: 12th IEEE International Conference on Peer-to-Peer Computing, P2P 2012, Tarragona, Spain, September 3-5, 2012, 2012, pp. 1–12.

[13] M. Mordacchini, L. Ricci, L. Ferrucci, M. Albano, R. Baraglia, Hivory: Range queries on hierarchical voronoi overlays, in: IEEE Tenth International Conference on Peer-to-Peer Computing, P2P 2010, Delft, The Netherlands, 25-27 August 2010, 2010, pp. 1–10.

[14] H. V. Jagadish, B. C. Ooi, Q. H. Vu, BATON: A balanced tree structure for peer-to-peer networks, in: Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, 2005, pp. 661–672.

[15] M. A. Arefin, M. Y. S. Uddin, I. Gupta, K. Nahrstedt, Q-tree: A multi-attribute based range query solution for tele-immersive framework, in: 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada, 2009, pp. 299–307.

[16] A. Datta, M. Hauswirth, R. John, R. Schmidt, K. Aberer, Range queries in trie-structured overlays, in: Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on, IEEE, 2005, pp. 57–66.

[17] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, S. Shenker, Prefix hash tree: An indexing data structure over distributed hash tables, in: Proc. of the 23rd ACM Symposium on Principles of Distributed Computing, 2004.

[18] D. Carfi, M. Coppola, D. Laforenza, L. Ricci, DDT: A distributed data structure for the support of P2P range query, in: 5th Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2009., IEEE, 2009, pp. 1–10.

[19] T. Pitoura, N. Ntarmos, P. Triantafillou, Saturn: range queries, load balancing and fault tolerance in dht data systems, Knowledge and Data Engineering, IEEE Transactions on 24 (7) (2012) 1313–1327.

[20] N. J. Navimipour, A. M. Rahmani, A. H. Navin, M. Hosseinzadeh, Resource discovery mechanisms in grid systems: A survey, Journal of Network and Computer Applications 41 (2014) 389–410.

[21] S. Kaune, T. Lauinger, A. Kovačević, K. Pussep, Embracing the peer next door: Proximity in kademlia, in: Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on, IEEE, 2008, pp. 343–350.

[22] P. Costa, D. Frey, Publish-subscribe tree maintenance over a dht, in: Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on, IEEE, 2005, pp. 414–420.

[23] R. Bhagwan, G. Varghese, G. M. Voelker, Cone: Augmenting DHTs to support distributed resource discovery, Department of Computer Science and Engineering, University of California, San Diego, 2003.

[24] M. Marzolla, M. Mordacchini, S. Orlando, Tree vector indexes: efficient range queries for dynamic content on peer-to-peer networks, in: 4th Euromicro Int. Conf. on Parallel, Distributed,and Network-Based Processing,PDP 2006., IEEE, 2006, pp. 8–pp.

[25] N. Shrivastava, C. Buragohain, D. Agrawal, S. Suri, Medians and beyond: new aggregation techniques for sensor networks, in: Proc. of the 2nd Int.Conf.on Embedded networked sensor systems, ACM, 2004, pp. 239–249.

[26] J. Lawder, The application of space-filling curves to the storage and retrieval of multi-dimensional data, Ph.D. thesis, Citeseer (2000).

[27] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.

[28] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, International Business Machines Company, 1966.

[29] T. Skopal, M. Krátkỳ, J. Pokornỳ, V. Snášel, A new range query algorithm for universal b-trees, Information Systems 31 (6) (2006) 489–511.

[30] K. Shudo, Y. Tanaka, S. Sekiguchi, Overlay weaver: An overlay construction toolkit, Computer Communications 31 (2) (2008) 402–412.

[31] Postbox location, `http://dracos.co.uk/made/locating-postboxes/export.php`, [].

[32] S. Berchtold, D. A. Keim, High-dimensional index structures database support for next decade's applications (tutorial), in: ACM SIGMOD Record, Vol. 27, ACM, 1998, p. 501.