HQ-Filter: Hierarchy-Aware Filter For Empty-Resulting Queries in Interactive Exploration

Akil Sevim
University of California, Riverside
Riverside, California
asevi006@ucr.edu

Ahmed Eldawy
University of California, Riverside
Riverside, California
eldawy@ucr.edu

***Abstract*—Modern visual data exploration systems are designed as client-server applications where the front-end interface generates a large number of queries to the back-end which are handled by a database server. As data exploration being a trial and error process, a significant amount of these queries return an empty result, which does not change the state of the visualization. These requests still add a significant overhead on network communication, request handling, and data processing. Moreover, given the virtually unlimited query space, it is impractical to enumerate and send all empty (or all non-empty) queries to the client to filter them. This paper introduces HQ-Filter, a hierarchy-aware filter for empty resulting queries, which utilizes the hierarchical nature of the data to construct a configurable and probabilistic filter. HQ-Filter can filter out empty-resulting queries at the client-side with a minimal size and processing overhead. HQ-Filter is applied to two existing data exploration systems for geospatial data, UCR-Star and Cloudberry. In both cases, it can successfully eliminate hundreds of queries per user which results in up-to 66% increase in server capacity by providing up to 15x speedup for average response time and up to 90% decrease in the server workload.**

***Index Terms*—data exploration, empty-resulting query filter**

I. Introduction

The increasing availability of big data triggers a growth in the interest for the data analytics. Data exploration is one of the key tasks in data analytics and many business intelligence and data exploration systems have been successfully deployed on a large scale such as Tableau, Qlik, and OmniSci.

In data exploration systems, users spend most of their time exploring, visualizing, and cleaning the data through a series of interactive queries [1]. As addressed in [2]–[5], a large number of these queries return an empty result. There are three reasons that cause this behavior and increase the number of the empty queries. First, data exploration is a trial and error process and users do not know the data distribution beforehand. Second, a single user action on the front-end can trigger in tens, and sometimes hundreds, of queries on the database to appropriately form a visualization possibly some parts of it is empty. Third, some of these systems generate many queries in the background to precache the result or to provide query recommendation [6], [7]. Data exploration systems also have different expectations on the response time. To keep the user active and engaged, the system must provide

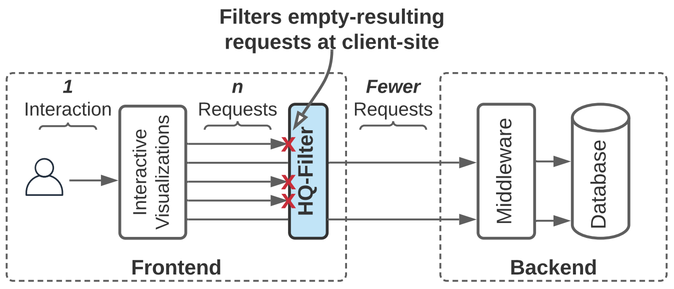


Fig. 1. HQ-Filter in an interactive data exploration system

a response time of around 500 milliseconds [8]. With the large number of queries and the fixed overhead of each query, e.g., parsing and optimization, a traditional DBMS will fall short in supporting this response time. In this paper, we argue that the best way to overcome this challenge is by reducing the number of queries sent to the database server.

The state of the art visual data exploration systems [9]–[12] support interactivity even for very large datasets by applying several methods such as caching, sampling, and pre-aggregation. Moreover, many systems proposed [13]–[17] to handle special needs of trajectory, mobile network, mobile sensor, and GPS data. However, none of these systems provide a solution to *avoid* the empty resulting queries.

This paper proposes HQ-Filter, a hierarchy-aware filter, which can reduce the number of queries sent from the client to the server in data exploration systems. Figure 1 shows the key idea of the proposed work. The server pre-constructs the HQ-Filter and ships it to the front-end. The client uses it to locally test and remove requests that will produce an empty result to offload the server-side processing. In that way, the user experience can be improved and the system would handle more concurrent users without disrupting the data exploration tasks. The challenge is to produce a compact filter that can be quickly shipped over network and can be efficiently processed at the client without adding a noticeable overhead.

The closest problem in literature is the empty-answer problem which results from very restrictive query constraints [18]. However, existing techniques in various domains [4], [19]–[25] focus on *query relaxation* which starts with an empty-answer query and modifies the constraints to avoid the empty

This work is supported in part by the National Science Foundation (NSF) under grants IIS-1954644, IIS-1838222 and CNS-1924694

result with the goal of providing an explanation for the user on why the result was empty. Unlike those methods, our problem focuses on detecting and pruning these queries with the goal of improving the system performance.

Besides query relaxation, [26] proposes a technique to store and reuse the atomic query parts that cause empty answers to detect the future queries that will return empty answer. However, this method relies on analyzing historical queries and works at the server-side. Applying it at the client-side will not be efficient for two reasons. First, each new user will start with an empty filter, so to have an efficient filtration, they have to use the system until the filter has enough history of queries. Our proposed filter does not require analyzing query history which makes it useful from the first query. Second, this method requires searching the historical atomic query parts which grows linearly with the filter size and might degrade the response time. Our proposed filter has a constant-time search algorithm which does not add any significant overhead.

In HQ-Filter, we utilize the natural hierarchy in the data to reduce the memory footprint of the filter so that it can be transferred over network and kept in the client main memory. Specifically, we build an approximate data structure based on Bloom Filters [27] that stores non-empty queries up-to a certain level in the data hierarchy and we use it to filter out as many empty queries as possible. The challenges in this approach are 1) ensuring a correct behavior of the application, i.e., we should not skip a non-empty query; and 2) selecting the optimal level in the data hierarchy to maximize the system performance.

We apply HQ-Filter in two real applications, UCR-Star [9] and Cloudberry [11]. In UCR-Star, the queries consist of map tile requests organized in a pyramid hierarchy of 20 zoom levels. In Cloudberry, the queries ask for the number of tweets in a specific region, state, county, or city, in a certain time interval. The data has two hierarchies, a geographical hierarchy and a time-based hierarchy. HQ-Filter can find the optimal level for each hierarchy to build the most efficient filter. We run an extensive experimental evaluation, which shows the efficiency of the HQ-Filter in improving the system performance with a negligible overhead on the client.

The contributions of this paper are summarized as follows:

- Formally define the problem of empty-resulting query filtering for hierarchical data.
- Propose HQ-Filter that can solve the hierarchical empty-resulting query filtering.
- Formulate an algorithm that can find the optimal hierarchy level to construct the HQ-Filter.
- Apply HQ-Filter in two real-world applications, UCR-Star and Cloudberry.
- Run an extensive experimental evaluation of HQ-Filter on real data.

The rest of this paper is organized as follows. Section II defines the problem. Section III describe the construction of the proposed HQ-Filter. Section IV provides an experimental evaluation of the system. Section V gives an overview of the related work. Finally, Section VI concludes the paper.

II. Problem definition

We first define the terms that we use in this study by considering an interactive data exploration system which uses the client-server architecture model.

Definition 1: (Resource) A piece of information that can be provided by the server. It can be a tile in a tiled map visualization, or a set of tweets along with various attributes as a JSON document for a spatiotemporal-textual data visualization.

A particular tile in tiled map visualization can be identified (Resource ID) with a unique reference represented as a tuple in a form of *(zoom_level,row,column)*. Similarly, a resource id for a tweet in a spatiotemporal-textual data visualization system can be *(geo_id, keyword, start_time, end_time)*. Clients send resource ids as requests to receive the resources as responses from servers.

Definition 2: (Empty Resource) A default resource which the server returns if the resource identifier does not match any resources. It can be an empty image tile for tiled map visualizations, or an empty set for a spatiotemporal-textual data exploration system.

Definition 3: (Empty-Resource Identifier) Any identifier for a resource that returns an empty result is called an empty-resource identifier.

At this point, we can define a preliminary version of the problem as follows. Given a limited memory size in bits, how to construct a data summary that can act as a filter to detect if a resource is empty given its identifier. The goal is to maximize the number of empty resources that can be correctly detected. We can easily show that an exact approach is infeasible due to the huge number of resources but we omit this part due to the limited space [28]. In the following part, we extend the definitions to introduce the notion of hierarchy in resources and use it to define our problem.

Definition 4: (Resource Hierarchy) A resource hierarchy is a logical organization of all resources in a tree structure with one root resource at the top. The terms *parent*, *child*, *ascendant*, and *descendant* are defined based on that logical tree structure. Note that the data covered by any child resource is a subset of the data covered by its parent.

Definition 5: (Resource Level) Each resource r has a non-negative integer level $l(r)$. By definition the root resource always has a level of zero. The level of any non-root resource r is defined as $l(r) = l(p) + 1$, where $l(p)$ is the level of its parent resource.

Definition 6: (Resource Cardinality $g(r)$) The cardinality of a resource r is the number of descendant resources under r. Formally, $g(r) = |\{r_2 : r_2 \text{ is a descendant of r}\}|$.

Definition 7: (Bounded Resource Cardinality $g(r, l_{max})$) The bounded cardinality of a resource r is the number of descendant resources under r that have a level of at most l_{max}. Formally,

$$g(r, l_{max}) = |\{r_2 : r_2 \text{ is a descendant of r} \wedge l(r_2) \leq l_{max}\}|$$

From the above definitions, we can observe that if a resource is empty, then all its descendants are also empty. This means

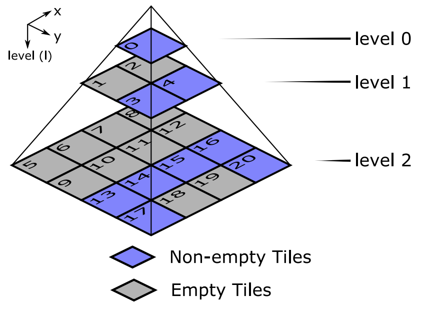


Fig. 2. Three levels of tiled visualization with empty and non-empty tiles

that if we have a filter that correctly detects a resource as empty with probability $1-p$, then we can use it to detect that all its descendants are also empty with the same probability.

Problem Definition: Given a set of resources R organized in a hierarchy with a maximum level of l_{max} and a memory budget m in bits, the problem is to construct a filter of at-most m bits that can detect the largest number of empty resources given their ID.

III. Hierarchy-Aware Filter For Empty-Resulting Queries (HQ-Filter)

In this section we introduce the Hierarchy-Aware Filter For Empty-Resulting Queries (HQ-Filter). The key idea of HQ-Filter is to build an probabilistic filter that stores only a subset of the non-empty resource IDs and utilize the data hierarchy to check for any resource ID. To further explain the key idea we present a toy example in Figure 2 which consists of 21 tiles organized in a hierarchy of three zoom levels with 9 non-empty tiles and 12 empty tiles. Let us assume that we want to build a filter of 8 bits. As a first cut solution we can build a Bloom Filter (BF1) with the 9 non-empty tiles (highlighted in blue). In a standard Bloom Filter, the optimal false positive rate will be $p_1 = 0.675$ which allows it to filter on average 3.9 out of the 12 empty tiles.

A better solution that utilizes the hierarchy is to build a Bloom Filter (BF2) for the three non-empty tiles at levels 0 and 1 which results in a false positive rate $p_2 = 0.278$. We say that $l_f = 1$ is the *filter level*. If a query asks for one of the tiles in levels 0 and 1, it can be tested directly from BF2. However, if a query requests a tile in level 2, then we check its parent in level 1. If the parent tile is empty, we conclude that the tile at level 2 must also be empty and we do not have to request it from the server. On the other hand, if the parent tile is not empty then we can only conclude that the tile in level 2 might be non-empty and we have to request it from the server. This adds another source of approximation; for example, tiles 18 and 19 will always be considered non-empty because their parents, tiles 3 and 4, respectively, are non-empty. On the other hand, tiles 5-12 will be detected as empty with probability $1-p_2$ since their parents will be queried in BF2. Therefore, the expected number of empty tiles that can be skipped in this approach is 7.22 as compared to 3.9 for BF1.

While this approach is simple to implement, the main challenge is how to efficiently determine the filter level (l_f) to achieve an overall optimal performance. Furthermore, if the data has multiple hierarchies, e.g., space and time hierarchies, then we need to find the optimal level for each of these hierarchies that together reach the optimal performance. In this paper, we formally define and solve the problem of constructing this filter optimally and we call it HQ-Filter thereafter. In the rest of this section, we first define a new metric that measures the quality of the filter for solving our problem and show how to find the filter with the highest quality. After that, we represent two case studies that show how to use HQ-Filter in real applications, namely, UCR-Star and Cloudberry. Lastly, we provide optimized counting techniques for resource IDs which is the most time-consuming step in the filter construction.

A. Performance metric

Our proposal is to utilize the observations that we explained in prior sections about the data hierarchy to build an efficient filter that detects empty resources up to a certain level l_f, and use the hierarchy to detect empty resources in deeper levels. In this part, we propose a performance metric P that can measure the performance of this filter. It is formally defined as the *expected* number of empty resources that the filter can correctly identify. The higher this value is, the more requests it can filter at the client-side. This metric is formally defined below.

$$P(l_f) = (1-p_f) \times \left(\sum_{l=0}^{l_f} |E_l| + \sum_{r \in E_{l_f}} g(r, l_{max}) \right) \tag{1}$$

Where,

- $P(l_f)$ is the expected number of empty resources that the filter can detect,
- l_f is the level of the HQ-Filter f,
- p_f is the false positive probability of the HQ-Filter f for resources at levels $[0, l_f]$,
- l is a level in the hierarchy, $l \in \{0, 1, 2, \dots, l_{max}\}$,
- E_l is the set of the empty resource identifiers at level l
- l_{max} is the maximum resource level a client can request,
- $g(r, l_{max})$ is the bounded resource cardinality, total number of descendants of a resource ID r up to level l_{max}

To compute p_f, we note that we internally build a BF for non-empty tiles in levels $0 \leq l \leq l_f$. Hence, we insert $n = \sum_{0 \leq l \leq l_f} |N_l|$ resource IDs into a filter with m bits. p_f can then be calculated according to the false positive probability equation $p = (1 - e^{-kn/m})^k$ [27].

The first summation in Equation 1 accounts for the empty resources that are inserted in the HQ-Filter. The second summation accounts for the empty resources that are descendants of an empty resource inserted in the HQ-Filter.

The optimal level l_f is the level that maximizes Equation 1. Therefore, to find the optimal level, we need to compute E_l and the function $g(r, l_{max})$ for levels $0 \leq l \leq l_{max}$. In the rest of this section, we first give a high-level algorithm that can find the optimal level and build the HQ-Filter and then we follow by concrete algorithms for two applications, UCR-Star and Cloudberry.

B. HQ-Filter Construction Overview

To construct an HQ-Filter for a dataset D, we propose an algorithm that consists of three steps, counting, optimization, and construction, described briefly below.

Step 1 - Counting: In this step, given a dataset D and a maximum depth l_{max}, we count the number of non-empty resources for all levels $0 \leq l \leq l_{max}$, i.e., $\langle |N_0|, |N_1|, \ldots, |N_{l_{max}}| \rangle$. l_{max} is the maximum depth of a resource that the user can query. For example, for tile-based visualization, most web maps support 20 zoom levels. There are three important points that we highlight in this step. First, while Equation 1 uses the number of empty resources per level $|E_l|$, we compute the non-empty resources since they are easier to compute as they are much fewer and then we compute E_l from it given the total number of resources per level. Second, we use a distributed process to count all the non-empty resources with one pass over the data as detailed shortly. Third, for some cases, it could be impractical to exactly count non-empty resources for all levels due to the excessive number of non-empty resources at the deep levels. In this case, we propose two alternatives in Section III-E that optimize the counting step.

Step 2 - Optimization: This step takes as input the non-empty resource counts $|N_*|$, the maximum level that the user can request l_{max}, and the size of the filter in bits m, and computes the optimal level at which the HQ-Filter should be constructed (l_f). Given that the total number of levels is usually small, e.g., tens of levels, we perform this step by applying Equation 1 for each level and choosing the best. The main challenge is to efficiently compute the bounded resource cardinality, $g(r, l_{max})$, as described in Definition 7.

Step 3 - HQ-Filter Construction: This final step takes as input the level to construct the HQ-Filter (l_f) and constructs the filter. It finds all the non-empty resource IDs at levels zero through l_f and inserts all of them into HQ-Filter. We show how to perform this step in a distributed environment on both Spark and AsterixDB.

C. HQ-Filter Construction for UCR-Star

In this section, we give the concrete algorithm that constructs HQ-Filter for the case of tiled map visualization as in UCR-Star [9]. To make a concrete algorithm, we have to do five steps. First, we define the resource ID and the hierarchy of the data. Second, we show how to efficiently count the number of non-empty resources per level. Third, we define the bounded resource cardinality $g(r, l_{max})$ to use in the optimization phase. Fourth, we show how to construct the filter once the level l_f is found. Finally, we explain how the client uses the filter to detect empty resources.

Tile Hierarchy: In the tiled map visualization, the resources comprise tiles organized in a quad-tree-like hierarchical structure. Each tile is identified by the triple (l, x, y), where $0 \leq l \leq l_{max}$ is the zoom level, and $0 \leq x, y < 2^l$ is the tile ID in that level. The root tile is $(0, 0, 0)$. For a non-root tile (l, x, y), the parent is the tile $(l-1, \lfloor x/2 \rfloor, \lfloor y/2 \rfloor)$. Each tile (l, x, y) has four children with IDs $(l+1, 2x+i, 2y+j)$ where $i \in \{0, 1\}$ and $j \in \{0, 1\}$.

Step 1 - Counting: To count the number of non-empty resources per level $|N_l|$, we run a distributed Spark job on the input data. In short, it partitions the data into 128 MB partitions, generates the set of IDs per level per partition in parallel, and finally combines them and counts the number of tiles per level. In more details, it first applies the `mapPartition` Spark transformation to build a set of non-empty resource IDs per level N_l, where $0 \leq l \leq l_{max}$. For each record in the input, it first finds the tile that contains the record at the deepest level l_{max}. Then, it adds that tile and all its ancestors in the hierarchy up-to the root tile to the corresponding sets of tile IDs per level. All these tile IDs N_l are stored as hash sets to ensure a constant-time process per insertion and avoid repetition. The output of this step is a set of pairs $\langle l, N_l \rangle$ for each level. These pairs are aggregated using the `reduceByKey` Spark transformation to produce one set of tile IDs per level which are finally counted. Even though all steps in this algorithm are parallelized, there is a bottleneck in merging the deepest level l_{max} which contains the largest number of non-empty tiles. We propose two optimizations in Section III-E when the N_l is extremely large.

Step 2 - Optimization: To find the optimum level l_f to construct the filter, the next step is to compute the performance P for each level in the multi-level visualization pyramid. To do that, we first define the bounded resource cardinality function g for this application. Due to the uniformity of the tile structure, where each tile has exactly four children, we propose the following function to compute g efficiently:

$$\begin{aligned} g(r, l_{max}) &= \sum_{l=l_r+1}^{l_{max}} (4^{(l_{max}-l)+1}) \\ &= \frac{4 \times (4^{l_{max}-l_r} - 1)}{3} \end{aligned} \tag{2}$$

As a result, when we apply the $g(\cdot, \cdot)$ for multi-level visualization to Equation 1, we can simplify the equation as follows.

$$P(l_f) = (1 - p_f) \times \left(\sum_{l=0}^{l_f} |E_l| + |E_{l_f}| \frac{4 \times (4^{l_{max}-l_f} - 1)}{3} \right) \tag{3}$$

Equation 3 is much more efficient than Equation 1 because it eliminates the second summation since all tiles at level l_f have the same bounded resource cardinality. Thus, we replace the summation with a multiplication by the number of empty resources at level l_f, i.e., $|E_{l_f}|$.

Now, we can compute P for all levels in the multi-level visualization pyramid to find the optimum level that maximizes P. For each level l_f, we calculate $n = \sum_{0 \leq l \leq l_f} |N_l|$ which allows us to compute p_f given the memory budget m. To calculate $|E_l|$, we observe that the total number of tiles in level l is 4^l. It directly follows that $|E_l| = 4^l - |N_l|$. After we have the above terms ready, we can easily use them in Equation 3 and find P_l for each level l to pick the optimum level l_f which maximizes P.

Step 3 - HQ-Filter Construction: The last step is to construct the HQ-Filter by inserting all of the non-empty RIDs $\{N_0, N_1, \ldots, N_{l_f}\}$ up to level l_f that we computed in the previous step. Similar to BF, we hash the RIDs using k hash functions, and set the corresponding indices to 1 among all m bits. Please recall that, each tile in multi-level visualization system is represented as (l, x, y) tuples. To have an efficient way of representing RIDs for tiles, we concatenate the bits of l, x, and y into a 64-bit long. Then by using the MurmurHash3 [29] and hashing method from [27], we hash the unique Tile IDs and set k number of bits to 1 in the bitarray.

Client-side Implementation: UCRStar's front-end is built as a web application using a tiled layer in OpenLayers. To integrate the HQ-Filter, the client requests the HQ-Filter from the server at startup and caches it in memory. Then, we intercept the tile request and test if the tile is empty. If the level of the requested tile is less than or equal to l_f, we directly test if the tile ID is in the filter. Otherwise, we look up the parent tile at level l_f which has the ID $(l_f, x \gg (l - l_f), y \gg (l - l_f))$, where $\gg$ is the logical right shift operation. If the tested tile is in the filter, the request is sent to the backend, otherwise, we skip the call and show an empty tile. This test runs in constant time which adds a negligible overhead on the client.

D. HQ-Filter for Spatiotemporal-Textual Data Visualization

Similar to the previous section, we focus on four parts. Based on how Cloudberry [11] works, the user searches for a keyword and all the queries contain this keyword. By design, Cloudberry builds a materialized view for the tweets that match the keyword. All the steps mentioned in this section are assumed to work on that materialized view. First, we define the hierarchy which, in this application, consists of two hierarchies for space and time. Second, we show how to count the non-empty resources in AsterixDB. Third, we calculate the bounded resource cardinality g. Fourth, we construct the HQ-Filter using a SQL++ query in AsterixDB.

Resource Hierarchy: In this case study, we have two hierarchies to deal with, spatial and temporal. The spatial hierarchy is defined by administrative levels, e.g., country, state, and city. The temporal hierarchy splits the entire time range into 1, 2, 4, 8, … partitions and so on. For the temporal hierarchy, we define the number of levels such that at the deepest level, each resource covers one day. In summary, we define two levels, one for each hierarchy, $0 \leq l_{geo} \leq l_{geo-max}$ where $l_{geo} = 0$ covers the entire input space; and the other for time $0 \leq l_{time} \leq l_{time-max}$. To combine them, we can define the level as a tuple $l = (l_{time}, l_{geo})$. However, to keep it simple, we assign a unique integer to each level so that $0 \leq l < (l_{geo-max} + 1)(l_{time-max} + 1)$. In this case, we can easily convert back and forth using the following equations:

$$l = (l_{geo-max} + 1) \cdot l_{time} + l_{geo} \tag{4}$$

$$l_{time} = \lfloor l / (l_{geo-max} + 1) \rfloor \tag{5}$$

$$l_{geo} = l \mod (l_{geo-max} + 1) \tag{6}$$

The linearization of the levels from multiple hierarchies makes it easier to apply Equation 1 in the optimization step. In the rest of this section, we will use l and (l_{time}, l_{geo}) interchangeably. In this design, each resource is identified by a pair $(geoID, timeID)$ where $geoID$ uniquely identifies a location, e.g., a state or a country, and $timeID$ identifies a single day or a range in the time hierarchy. The level of a resource r is $l_r = (l_{r-time}, l_{r-geo})$

Step 1 - Counting: We count the number of non-empty resources in the input data using a SQL++ query that runs in AsterixDB. To do that, we need to calculate the $timeID$ and $geoID$ for each tweet at each level. Since the $timeID$ has a regular structure, we implement a user-defined function (UDF) that takes the tweet timestamp and the level l_{time} and returns the $timeID$. For the geoID, we run a spatial join operation between the tweet geolocation ($longitude$, $latitude$) and the boundaries of the geographical regions, e.g., countries and states, and project the geoID of each level as an additional column in the data table. To count the number of non-empty resources per level, we perform one grouped aggregation SQL++ query that groups the tweets by level and counts the distinct IDs per level.

Step 2 - Optimization: To run the optimization step, this part defines the bounded resource cardinality $g(r, l_{max})$. Given the irregularity of the spatial hierarchy, we build a lookup table $s(geoID)$ which contains the total number of spatial resource IDs that are descendants of $geoID$. For example $s(US)$ contains the total number of states and cities in the US, while $s(NY)$ contains the total number of cities in New York state. Hence, we define g as follows:

$$\begin{aligned} g(r, l_{max}) &= \sum_{l=l_{r-time}+1}^{l_{time-max}} (2^{l_{time-max}-l}) \cdot s(r.geoID) \\ &= 2^{l_{time-max}-l_{r-time}} \cdot s(r.geoID) \end{aligned}$$

According to the above definition, the bounded resource cardinality can be computed in constant time. However, unlike the tiled map application, tiles at the same level do not all have the same cardinality. Hence, we still need to iterate over each empty resource at level l_f while calculating the filter performance using Equation 1. If the number of empty resources is very large, we can instead calculate the summation of the *non-empty* resources and subtract that from the total number of resources in levels l_f+1 to l_{max} which is a constant that is independent of the empty and non-empty resource IDs.

Step 3 - Construction: Once the optimal level l_f is selected, this step constructs the HQ-Filter by inserting all the non-empty resources in levels zero through l_f. We implement this step as a SQL++ query in AsterixDB. To do that, we define a new UDF that computes the hash function from the resource ID $h(geoID, timeID)$. We apply this function to select the bit positions in the HQ-Filter that need to be set. Then, we iterate over all these bit positions and set them to construct HQ-Filter. Notice that regardless of the data set, the number of bits to set is bounded by the filter size.

Client-side Implementation: Cloudberry [11]'s front-end is implemented as a web application which uses a vector layer to display a choropleth map. When the user enters a keyword, the client requests the corresponding HQ-Filter. As the user navigates the map, the system generates a sequence of requests for each visible region and every day. The request level (l_{geo}, l_{time}) is compared to the HQ-Filter level l_f. If both are less than or equal their corresponding filter level l_f, we directly test if the resource is non-empty in the filter. Otherwise, we locate the ascendant resource at level l_f. To find that resource in the geospatial hierarchy, the client keeps a lookup table that contains the ascendant geoID at level l_f for each resource. The parent resource in the time hierarchy can be easily obtained using simple calculations similar to the one used in the tiled map case study.

E. Optimizations for Counting

For both case studies described earlier, counting is the most expensive step for two reasons. First, it counts the number of non-empty resource IDs in *all levels*. Second, depending on the density of the dataset, the number of non-empty resource IDs could be tremendously large in deep levels. For example, in the `Parks` dataset which has only 10 million polygons, the number of non-empty resources in level 19 is around 11 billion.

We make two observations that we utilize in this section to speed up the counting process. First, since the goal of the counting step is to find the optimal level in Step 2, we can use approximate counts and still get the same result. The first optimization uses approximate counting to obtain the optimal levels l_f. Second, based on the behavior of the performance function P in Equation 1, we can find the optimum level l_f by counting the number of non-empty resource IDs in a few levels around the optimal level l_f. However, since l_f is unknown, the second optimization uses incremental counting to search for the optimal level but without counting all the levels.

1) Approximate Counting: Since the goal of the counting step is to find the optimal level l_f, an estimation of these resource counts can still produce an accurate answer. Therefore, exact counting might be unnecessary. The first proposed optimization is to replace exact counting with approximate counting per level. We can control the desired accuracy so that we can still achieve the same result for the overall algorithm, i.e., find the same optimum level l_f.

The approximate counting algorithm uses the HyperLogLog approximate counting algorithm [30] which is widely used and already supported by some big data frameworks, e.g., Spark. We used Streamlib's implementation of HyperLogLog++ (HLL++). Given that all the problematic cases that we faced for counting were in the tiled map visualization application, we only implemented this approach in Spark but the main idea can easily apply to other systems. The implementation is similar to the exact counting algorithm described in Section III-C. However, we replace the partial and full hash sets with the HLL counting structure. This means that each worker approximately counts the number of unique resource IDs for each level using HLL. Then, the partial HLL structures for each level are transferred to one machine which combines them to compute the final approximate count. The remaining steps work exactly as before but they use the approximate counts instead of the exact ones.

2) Incremental Counting: This part shows a second optimization which relies on the properties of the optimization function P. We observe that the function has only one global maximum and no suboptimal local maxima. This means that if we find any local maximum, we can directly use it as the global maximum as well. This optimization combines the counting and the optimization steps into one step to count and search for the maximum, simultaneously. The way it works is that it starts by counting the top few levels, e.g., seven levels, since they are cheap to compute anyway. If a local maximum is found, it is returned. Otherwise, it incrementally counts one additional level at each iteration and tests if a local maximum has been found. Once the performance $P(l_f)$ starts to decrease, the algorithm stops and returns the level that produced the highest performance.

Notice that the two optimizations are orthogonal which means we can apply either or both of them if needed.

IV. Experiments

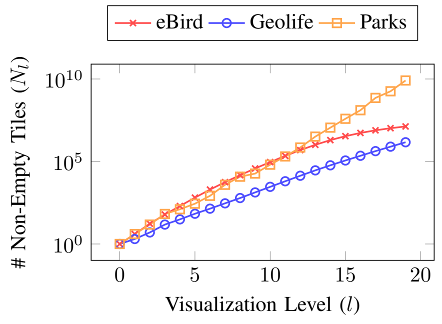
We evaluate the HQ-Filter applied in two real-world applications: UCRStar [9] and CloudBerry [11]. As suggested in [2], our experiments have been conducted with real-world data and workload. We also provide a simplistic user behaviour categorization and data analysis to highlight the empty-answer aspects of both data and workload. Then, we measure the empty resulting query detection performance of HQ-Filter, average response time of the data exploration systems and the capacity of maximum concurrent users that can use the system within interactivity limits.

A. Setup

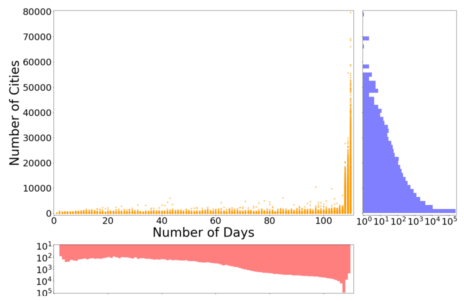
1) Software and Hardware setup: We have three baselines. First, No-filter, when all the requests are sent to the server. Second, Bloom Filter (BF), when all non-empty resources are added to a regular BF. Third, the detection technique from [26] (LUO). We implement the filter by following the original design, and construct the filter by adding queries from 30 users. Then, we evaluate the filter with the test group.

All experiments have been conducted on Amazon Web Services (AWS). We used **m5.xlarge** instance type which has 4 vCores, 16 GB memory. HQ-Filter creation experiments run on an a cluster with 20 machines each has 200 GB SSD.

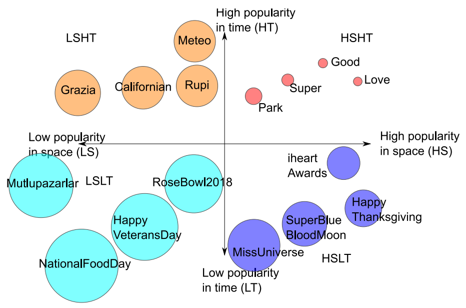
2) Datasets: Table I lists all **datasets** that we use. The first three are used with tiled map visualization and are available at UCR-Star [9]. The fourth is used with spatiotemporal-textual



(a) Tiled Map Datasets



(b) Spatial and temporal distributions of keywords



(c) Sample keywords for the four categories

Fig. 3. Analysis of the datasets used in the experiments

TABLE I
DATASETS

Name	Size	Records	Description
eBird	211.2 GB	566 Million	Points
Parks	7.9 GB	10 Million	Polygons
GeoLife	1.7 GB	23 Million	Points
Tweets	1.6 TB	387 Million	Geo-tagged Tweets

data exploration. Figure 3(a) shows the number of non-empty tiles per level for the datasets used with tiled map visualization which highlights the characteristics of these datasets.

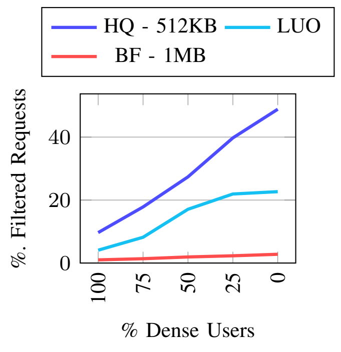
To choose a representative set of keywords from the Twitter dataset, we first analyze the keywords as illustrated in Figure 3(b). For each keyword, we count the number of empty days and number of empty cities to measure its temporal and spatial popularity, respectively. We use the average along both dimensions to split the keywords into four categories based on their popularity in time (LT/HT) and space (LS/HS). Figure 3(c) shows the keywords that we picked to represent each category. The radius of each bubble indicates the total number of empty resources which is a good indicator for the *potential saving* of empty resulting queries for this keyword.

3) Workload: Based on each dataset/keyword characteristics, we define two types of users, *sparse* and *dense*. Sparse users are those interested in regions that have a lot of empty resources while dense users are the opposite. We pre-record several timed sequences of requests for both user types and use them as an input workload which provide the ability to control the ratio of dense and sparse users.

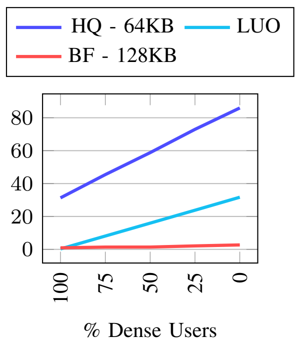
4) Parameters & Metrics: We vary the filter size (m), dataset size, ratio of dense users, and number of concurrent users as our parameters. For our evaluations, we measure the HQ-Filter creation time, average response time per request, server workload, and the number of filtered requests.

B. Evaluation of Detection Performance

In this experiment we evaluate the empty resulting query detection performance of the HQ-Filter and compare it to the baselines. Our analysis also provide insights about how the detection performance is affected by the user types. Recall that *sparse* (*dense*) users who are interested in areas with large (small) number of empty resources.



(a) eBird



(b) HSHT keywords

Fig. 4. Percentage of the filtered requests as the ratio of dense users changes

1) Tiled map visualization: In Figure 4(a), we vary the percentage of dense users from 100% (all users are dense) to 0% (all users are sparse) and measure the percentage of requests that the filter can detect, i.e., higher is better. We first observe, HQ-Filter is consistently better even when it occupies half the size of BF. This is a direct result from accounting for the data hierarchy which allows it detect more empty resources with less size. Second, HQ-Filter outperforms the third baseline, and the performance of it is not consistent since the detection rate is highly dependent on previous user data. Lastly, the less the dense users, the better the filters behave since there is a larger pool of empty requests to filter. This experiment also reveals that BF is not as performant as HQ-Filter due to its false-positive rate approaching 1.0 even with a memory budget of 1MB. A memory budget larger than 1MB is not practical for the proposed problem since it gets computed on the server and transferred over network to the client. Other datasets produce a similar behavior but we omit the results due to the limited space. Interested readers can refer to the technical report [28].

2) Spatiotemporal-Textual Data Exploration: We evaluate the empty answer detection performance of HQ-Filter when integrated into the Cloudberry. Please note that, in this use case, we create a filter for each keyword. Thus, we have a small number of elements to insert into the filters and our filters should be small enough to be kept in client side for multiple keywords. Figure 4(b) reports the percentages of the

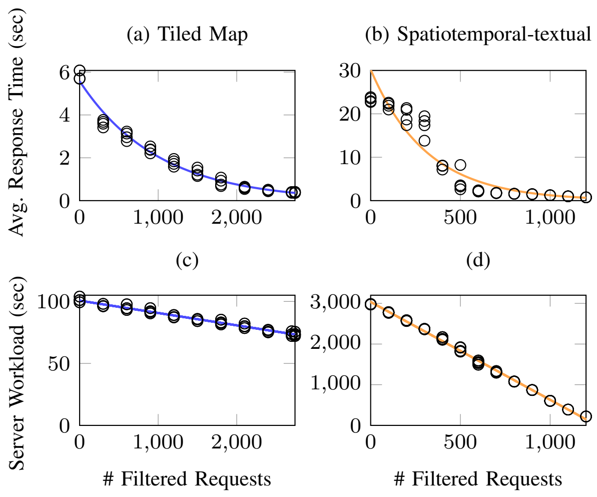


Fig. 5. The effect of filtering empty requests on the client response time and server workload

filtered empty resulting request while we vary the percentage of dense users. Here we report the results for HSHT keywords. The other keyword categories provide a similar behavior as detailed in [28]. These experiments show that HQ-Filter has a better empty resulting query detection performance over the other baselines. Similar to tiled map visualization, the less dense users we have, the better the filters behave due to the availability of more empty resources to skip. Please note that for each keyword, the workload contains users who zoom into different countries with different languages. As a result, most of the requested resources for sparse users return empty results which is the reason the percentage of filtered requests is relatively high. For the same reason, third baseline's performance falls behind HQ-Filter since the previous users queries are helping to the future queries only for country level.

C. Response Time Evaluations

We can effectively filter the empty queries at the client side for both use cases but it is crucial to show that doing this improves system performance. Figure 5 shows the effect of eliminating empty-resulting queries on both the client and server performance. We vary the number of filtered requests in a controlled way to accurately measure its affect and we repeat each point four times for accurate results. We use the eBird dataset for the tiled map, and the 'love' keyword for and spaitotemporal-textual exploration. The workload consists of eight dense and eight sparse concurrent users.

1) Average Response Time (client-side): as we filter more empty resource, the average response time reduces significantly. The correlation coefficient for the average response time and the number of filtered empty requests is -0.923 and -0.896 for the tiled map and the spatiotemporal-textual data exploration, respectively. This is a result of saving the transmission of the request over network, the processing on the server side, and the queuing of requests at the server. It is worth mentioning that the times converge to a fixed value

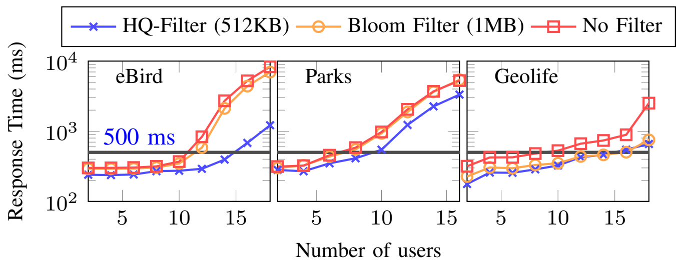


Fig. 6. Server performance after filtering empty requests

which represents the time needed to process the non-empty requests since the filter only saves the empty requests.

2) Server Workload: The figures Figure 5(c) and (d) show the effect of filtering the requests on the server side. The server workload is measured as the total time needed to handle all the requests. We see a similar behavior in which the server workload reduces significantly as we reduce the number of empty queries. Unlike the average response times, we see that the server workload is decreasing linearly since we measure the total time not the average. Additionally, the network round-trip time is not observed on the server side. Finally, we can observe that the performance of the spatiotemporal-textual exploration application improves significantly due to the high volume of requests that is typically sent by its front-end design.

The above experiments reveal the strong correlation between the number of filtered requests and the performance observed on both the client and server. For the rest of the experiments, we will focus on reporting the number of filtered requests since these experiments are easier to reproduce as they do not depend on the system load or hardware specification.

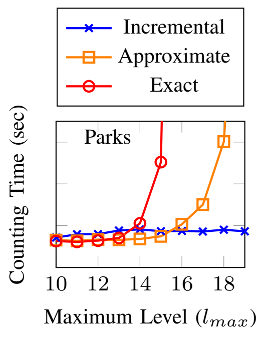
D. Improvement on server capacity

This experiment studies how the use of HQ-Filter can increase the server capacity in terms of number of concurrent users that can be supported while providing a processing time of less than 500 milliseconds. Figure 6 shows the average response time as the number of concurrent users increases from 1 to 20. We measure the performance of three approaches, *No Filter* is when all requests are handled, *BF* is when a Bloom filter is used to filter empty requests, and when HQ-Filter is used. In this experiment BF uses 1MB while HQ-Filter uses only 512KB of memory. The horizontal line indicates the cutoff response time of 500 milliseconds.

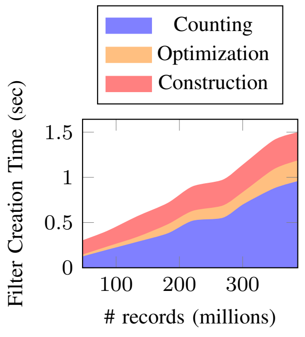
The experiment clearly shows that the use of HQ-Filter increases the capacity of the server for all three datasets. The server capacity increases by 50%, 66%, and 60%, for eBird, Parks, and Geolife datasets, respectively. While BF is close to the performance of HQ-Filter for the Geolife datasets, note that BF uses 1MB of memory while HQ-Filter uses only 512KB. Finally, note that this improvement of the server capacity does not require any change in the server architecture or design since all the filtering happens on the client side.

E. HQ-Filter Creation

In this section, we provide our evaluations of creating HQ-Filter for the two use-cases that we study.



(a) Counting non-empty tiles (zoom levels 10 to 19)



(b) For spatiotemporal-textual data application in AsterixDB

Fig. 7. Cost of HQ-Filter Creation

1) Tiled Map Visualization: In this experiment, we focus on the counting step which is the main bottleneck of the HQ-Filter creation process [28]. As Figure 3(a) shows, the number of non-empty resources can increase excessively for deep levels which makes this step very expensive for some datasets.

We report the time for the counting step of the three techniques as we increase l_{max} in Figure 7(a). For the Parks dataset, we see an interesting behavior where both the exact and approximate techniques take too long which makes both of them unpractical to use. This is due to the huge number of non-empty resources in deep levels. The approximate technique *is* faster but it still takes too long for deeper levels. The incremental technique shines for this case since it does not have to count the deeper levels. For the Parks dataset, the optimal level is 12 so the running time stabilizes after that point. For eBird and Geolife datasets, all three approach performs similarly because of these datasets have a reasonable amount of non-empty tiles for even deepest level which result in any of the approach performing well [28].

2) Spatiotemporal-Textual Data Visualisation: Figure 7(b) shows the breakdown of the HQ-Filter construction process in the spatiotemporal-textual data exploration application. Following the design of Cloudberry [11], the filter is created from a materialized view of the tweets that match the query keyword. We do not show the time for creating the materialized view since it is not part of the HQ-Filter creation process. Similar to the previous experiment, the counting step is the most expensive step. Since we keep the resource ids in memory after counting, the construction step remains stable. The optimization step for this application is slightly costly due to the complexity of the performance function P which needs to iterate over all non-empty resources and use the lookup table to count the number of descendent resources. The overall creation time is very fast with no more than 1.5 seconds for the largest dataset due to the use of materialized view. Hence, we did not implement the two optimized counting techniques, approximate and incremental, but they can be implemented in the same way if needed.

V. Related Work

This section summarizes the related work in data exploration into three categories, presentation, application, or data layer [1] by their focus.

Presentation layer represents the work that focus on providing an interactive interface that allows users to explore data effectively. This includes systems that use pen and touch devices [12], interactive map-based exploration [11], commercial business intelligence (BI) systems based on the grammar of graphics [31], [32], and JavaScript libraries for visualization [33]. [34] is focused on reasoning whether the system should make an empty resulting check and notify the user by taking account the psychological factors for visual query builders. All this work helps in building interactive exploration systems that encourages more users to explore the data. HQ-Filter can assist all these systems by filtering empty-resulting queries.

Application layer takes a user query from the presentation layer and processes it to return the result. Typically, processing a user query involves retrieving and querying data from the database layer. A prime example is Cloudberry [11] which breaks down a user query into several queries [35] sent to AsterixDB for processing. Some applications follow the *Approximate Query Processing (AQP)* approach [36]–[38] in which they use sampling to speed up the query processing of big data. While HQ-Filter is a probabilistic technique that detects empty queries, the final result is still exact since it only skips queries that are certainly empty. Our work can be combined with the work listed above where HQ-Filter is used to skip empty resulting queries, and these techniques are used to answer non-empty resulting queries.

Query relaxation is the problem of explaining why an SQL query produces an empty result [18]. The techniques in this topic [4], [19]–[25] can suggest alternative queries or *relaxations* of the conditions by relying on application customized optimizations. In contrast, the proposed approach does not try to modify the query or suggest an alternative but it just aims at quickly detecting queries that return an empty answer at the client-side.

Database layer stores and indexes the data to serve interactive exploratory systems. This work includes pushing visualization queries into the database [39], using a specialized query language for visualization [31], [40], and the use of distributed query engines [41], [42] and adaptive indexes [43], [44] for tiled map visualization. To improve the performance, some techniques use in-memory preaggregated indexes [45], WebGL [46], or sampling methods [47], [48]. GloBiMaps [49] proposes a randomized data structure that models sparse binary images, e.g., land and water, and provides an efficient query to find the value of a pixel as zero or one. It differs from our work by being a visualization method taking advantage of the empty areas in the geospatial datasets while our work focuses on pruning any empty-resulting query and is not limited to raster images. [26] proposes a method that analyzes previous SQL queries and extracts the *atomic* parts that can be used to detect empty results in future queries. HQ-Filter does not require past queries to be constructed and can be checked in constant time which make it suitable for client-side filtering. In summary, HQ-Filter can be applied at the client-side as an additional light-weight filter in addition to more advanced filters at the server-side that might need more information that is only available to the database server.

VI. Conclusion

In this paper, we showed that in data exploration systems, a large number of requests return an empty result. We defined the problem of detecting empty queries at the client side by utilizing the hierarchy in the data. We proposed HQ-Filter, as an efficient solution that uses a compact representation to detect empty queries at the client-side with minimal overhead in processing. We implemented HQ-Filter in two real applications and showed that it can effectively be implemented in both scenarios. Our extensive experimental evaluation showed that HQ-Filter can accurately eliminate most of the empty queries in both applications which results in up-to 66% increase in server capacity, provides up to 15x speedup for average response time, and up to 90% decrease in the server workload.

References

[1] S. Idreos, O. Papaemmanouil, and S. Chaudhuri, "Overview of data exploration techniques," in *SIGMOD*, 2015.

[2] P. Eichmann, E. Zgraggen, C. Binnig, and T. Kraska, "Idebench: A benchmark for interactive data exploration," in *SIGMOD*, 2020.

[3] H. Guo *et al.*, "A case study using visualization interaction logs and insight metrics to understand how analysts arrive at insights," *TVCG*, vol. 22, no. 1, pp. 51–60, 2016.

[4] I. Dellal *et al.*, "On addressing the empty answer problem in uncertain knowledge bases," in *DEXA*, 2017.

[5] M. L. Kersten *et al.*, "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds," *PVLDB*, vol. 4, no. 12, pp. 1474–1477, 2011.

[6] L. Battle, R. Chang, and M. Stonebraker, "Dynamic prefetching of data tiles for interactive visualization," in *SIGMOD*, 2016.

[7] K. Wongsuphasawat *et al.*, "Towards a general-purpose query language for visualization recommendation," in *HILDA*, 2016.

[8] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *TVCG*, vol. 20, no. 12, pp. 2122–2131, 2014.

[9] S. Ghosh *et al.*, "UCR-STAR: The UCR Spatio-Temporal Active Repository," *SIGSPATIAL Special*, December 2019.

[10] M. Sarwat, "Interactive and scalable exploration of big spatial data - A data management perspective," in *MDM*, 2015.

[11] S. Su *et al.*, "Visually analyzing A billion tweets: An application for collaborative visual analytics on large high-resolution display," in *IEEE BigData*. Seattle, WA: IEEE, Dec. 2018, pp. 3597–3606.

[12] A. Crotty *et al.*, "Vizdom: Interactive analytics through pen and touch," *PVLDB*, vol. 8, no. 12, pp. 2024–2027, 2015.

[13] A. M. Hendawi *et al.*, "An interactive map-based system for visually exploring and cleaning GPS traces," in *SIGSPATIAL*. ACM, 2019.

[14] X. Ding *et al.*, "VIPTRA: visualization and interactive processing on big trajectory data," in *MDM*, 2018.

[15] M. Dash *et al.*, "An interactive analytics tool for understanding location semantics and mobility of users using mobile network data," in *MDM*, 2014.

[16] A. Sawas *et al.*, "Trajectolizer: Interactive analysis and exploration of trajectory group dynamics," in *MDM*, 2018.

[17] J. Liono *et al.*, "UTE: A ubiquitous data exploration platform for mobile sensing experiments," in *MDM*, 2016.

[18] S. Gathani, P. Lim, and L. Battle, "Debugging database queries: A survey of tools, techniques, and users," in *CHI*, 2020.

[19] X. Lyu and W. Hu, "RQE: rule-driven query expansion to solve empty answers in SPARQL," in *JIST*, 2019.

[20] M. Wang *et al.*, "Towards empty answers in SPARQL: approximating querying with RDF embedding," in *ISWC*, 2018.

[21] D. Mottin *et al.*, "A holistic and principled approach for the empty-answer problem," *VLDB*, 2016.

[22] E. Vasilyeva *et al.*, "Answering "why empty?" and "why so many?" queries in graph databases," *J. Comput. Syst. Sci.*, 2016.

[23] E. Vasilyeva, T. Heinze, M. Thiele, and W. Lehner, "Debeaq - debugging empty-answer queries on large data graphs," in *ICDE*, Helsinki, Finland, May 2016, pp. 1402–1405.

[24] D. Mottin *et al.*, "IQR: an interactive query relaxation system for the empty-answer problem," in *SIGMOD*. Snowbird, UT: ACM, Jun. 2014, pp. 1095–1098.

[25] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis, "A probabilistic optimization framework for the empty-answer problem," *PVLDB*, vol. 6, no. 14, pp. 1762–1773, 2013.

[26] G. Luo, "Efficient detection of empty-result queries," in *VLDB*, 2006.

[27] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, p. 187–218, Sep. 2008.

[28] A. Sevim and A. Eldawy, "Hq-filter: Hierarchy-aware filter forempty-resulting queries in interactive exploration," 2021. [Online]. Available: https://www.cs.ucr.edu/ eldawy/publications/21-HQ-Filter-Technical-Report.pdf

[29] A. Appleby, "Murmurhash3," 2016. [Online]. Available: https://github.com/aappleby/smhasher/wiki/MurmurHash3

[30] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *EDBT*. Genoa, Italy: ACM, Mar. 2013, pp. 683–692.

[31] C. Stolte and P. Hanrahan, "Polaris: A system for query, analysis and visualization of multi-dimensional relational databases," in *INFOVIS*. Sald Lake City: IEEE Computer Society, Oct. 2000, pp. 5–14.

[32] L. Wilkinson, *The Grammar of Graphics, Second Edition*, ser. Statistics and computing. Springer, 2005.

[33] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *TVCG*, vol. 17, no. 12, pp. 2301–2309, 2011.

[34] S. S. Bhowmick *et al.*, "Interruption-sensitive empty result feedback: Rethinking the visual query feedback paradigm for semistructured data," in *CIKM*, 2015.

[35] J. Jia, C. Li, and M. J. Carey, "Drum: A rhythmic approach to interactive analytics on large data," in *IEEE BigData*, Boston, MA, Dec. 2017.

[36] K. Li and G. Li, "Approximate query processing: What is new and where to go? - A survey on approximate query processing," *Data Sci. Eng.*, vol. 3, no. 4, pp. 379–397, 2018.

[37] J. R. Rojas *et al.*, "Sampling techniques to improve big data exploration," in *LDAV*, 2017.

[38] A. B. Siddique, A. Eldawy, and V. Hristidis, "Comparing synopsis techniques for approximate spatial data analysis," *PVLDB*, vol. 12, no. 11, pp. 1583–1596, 2019.

[39] E. Wu, L. Battle, and S. Madden, "The case for data visualization management systems," *PVLDB*, vol. 7, no. 10, pp. 903–906, 2014.

[40] R. M. G. Wesley, M. Eldridge, and P. Terlecki, "An analytic data engine for visualization in tableau," in *SIGMOD*, 2011.

[41] A. Eldawy, M. F. Mokbel, and C. Jonathan, "HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data," in *ICDE*, 2016.

[42] J. Yu, Z. Zhang, and M. Sarwat, "Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem," in *SSDBM*, 2018.

[43] S. Ghosh, A. Eldawy, and S. Jais, "AID: an adaptive image data index for interactive multilevel visualization," in *ICDE*, 2019.

[44] S. Ghosh and A. Eldawy, "AID*: A Spatial Index for Visual Exploration of Geo-Spatial Data," 2020.

[45] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *TVCG*, vol. 19, no. 12.

[46] Z. Liu, B. Jiang, and J. Heer, "*imMens*: Real-time visual querying of big data," *Comput. Graph. Forum*, vol. 32, no. 3, pp. 421–430, 2013.

[47] Y. Park, M. J. Cafarella, and B. Mozafari, "Visualization-aware sampling for very large databases," in *ICDE*, 2016.

[48] J. Yu and M. Sarwat, "Turbocharging geospatial visualization dashboards via a materialized sampling cube approach," in *ICDE*, 2020.

[49] M. Werner, "Globimaps - A probabilistic data structure for in-memory processing of global raster datasets," in *SIGSPATIAL*, 2019.