UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Query Processing Techniques for Data Exploration in Heterogeneous and Distributed Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Akil Sevim

September 2023

Acknowledgments

I deeply appreciate the support and guidance from Dr. Ahmed Eldawy during my thesis work. His exceptional teaching and emphasis on progress over mere results have greatly enriched my academic journey. I'm fortunate to have learned and grown under his guidance. Additionally, I'm deeply grateful to my committee members, Dr. Vassilis Tsotras, Dr. Vagelis Hristidis, and Dr. Amr Magdy, for their valuable insights and feedback, which significantly improved my work

I also want to express my sincere gratitude to Dr. Michael Carey. It has been a privilege to work alongside him, and his feedback and comments on my work have been truly enlightening. I also owe a great deal to the members of the Big Data Lab. To my fellow lab mates who have graduated, Dr. Tin Vu, Dr. Saheli Ghosh, and Dr. Samriddhi Singla, your warm welcome and continuous support from the beginning were instrumental. To the current members of the lab, Xin Zhang, Zhuocheng Shang, Majid Saeedan, and Tomal Majumder, I extend my gratitude for the countless enlightening discussions and collaborative efforts.

The text of this dissertation, in part, is a reprint of the material as it appears in 22nd IEEE International Conference on Mobile Data Management, 2021. The co-author Dr.Ahmed Eldawy listed in that publication directed and supervised the research which forms the basis for this dissertation.

To my parents, brother and family for all the support.

ABSTRACT OF THE DISSERTATION

Efficient Query Processing Techniques for Data Exploration in Heterogeneous and Distributed Systems

by

Akil Sevim

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2023
Dr. Ahmed Eldawy, Chairperson

The rise in the variety and amount of big data has sparked interest in data-driven applications. As a result, there is a growing demand for effective ways to explore, transform, and understand data across various platforms, including distributed systems. Getting insights from data involves cleaning, changing, showing, and combining data, which requires scalable systems for quick knowledge extraction. This thesis introduces new techniques for processing queries in big data management systems.

Data exploration involves trial and error, often yielding empty query results. To tackle this, HQ-Filter is introduced—an agile hierarchy-aware data structure. HQ-Filter exploits data hierarchy to build a configurable, probabilistic filter, efficiently eliminating empty-result queries on the client side. Applied to UCR-Star and Cloudberry systems for spatiotemporal-textual data, HQ-Filter significantly boosts server capacity (up to 66%), accelerates response times (up to 15x), and reduces server workload (up to 90%).

Moreover, for today's data scientists, combining diverse big datasets via distributed systems using join queries with complex conditions is essential. However, the availability of methods that can generate an optimized query plan for such queries in Database Management Systems (DBMSs) is limited due to the implementation and integration complexities. To overcome this issue, we introduce the Flexible User-defined Distributed Joins (FUDJ) framework, which seeks to enhance the availability of optimized join algorithms within DBMSs.

FUDJ enables partition-based distributed join algorithms without deep DBMS or distributed programming knowledge. Through a novel extensibility architecture, FUDJ enhances availability and diversity of optimized join algorithms, amplifying options for data scientists and database researchers.

FUDJ facilitates query processing by embedding it in any query optimizer. Using "CREATE JOIN," FUDJ deploys join libraries, detects flexible distributed join queries, constructs optimized plans, and offers execution options. Implemented in Apache AsterixDB, FUDJ delivers substantial efficiency gains (20x less work) and speedups (up to 1200x) compared to built-in and on-top approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's data-driven landscape, the need for efficient data analysis and exploration is more critical than ever. This demand is fueled by the exponential growth of data sources, ranging from social networks to IoT devices, and the rise of data-driven applications. As a result, the traditional approach to joining datasets in Database Management Systems (DBMS) is facing new challenges. Historically, DBMS treated "join" as a straightforward operation on structured data with simple equality conditions. However, the landscape has evolved, and data scientists now require the ability to combine large, diverse datasets from various sources, often involving complex conditions and less structured data.

This evolution has given rise to various types of join operations, especially in distributed systems, where data integration is a complex task. Existing methods for implementing new join operators can be categorized into three approaches: standalone, within distributed systems like Spark and Hadoop, and as built-in operators within DBMS. While each approach has its merits, they also come with limitations, such as inflexibility, nonadaptability, and performance concerns. Furthermore, integrating new join algorithms into DBMSs remains a challenge, leaving complex join queries to be handled by on-top solutions with User-defined Functions (UDFs) and Nested Loop Join (NLJ) operators, which can lead to slower query execution times.

On the other front, the surge in big data and data-driven applications has transformed the landscape of data exploration and business intelligence. Systems like Tableau, Qlik, and OmniSci have achieved significant success, generating substantial revenue in the realm of big data and business analytics. However, this growth has brought new challenges, particularly in the realm of data exploration systems.

Data exploration systems differ significantly from traditional DBMS workloads, as users often engage in trial-and-error processes, generating a substantial number of queries. Unfortunately, many of these queries result in empty outcomes, for three key reasons: the trial-and-error nature of data exploration, the proliferation of queries stemming from single user actions, and the background query generation for caching and recommendation purposes. These systems also require low response times to maintain user engagement, adding another layer of complexity.

To address these challenges, our work introduces HQ-Filter, which leverages the inherent hierarchy of data to efficiently filter out empty-resulting queries in visual data exploration systems. HQ-Filter's innovative approach employs an approximate data structure based on Bloom Filters, storing non-empty queries up to a certain level in the data hierarchy. This enables the system to quickly detect and discard empty queries, significantly reducing the query workload on the server. By providing a constant-time mechanism for detecting empty-result queries and effectively utilizing client-side resources, HQ-Filter enhances the performance of data exploration systems, ultimately facilitating more efficient and engaging data analysis.

In addition to HQ-Filter, we present the Flexible User-defined Distributed Joins (FUDJ) approach, which aims to revolutionize the landscape of join algorithms within DBMS. As data scientists grapple with the complexities of integrating and optimizing join operations for diverse datasets, FUDJ offers a user-friendly solution. It empowers users with varying levels of expertise to efficiently leverage join algorithms, significantly reducing the code and knowledge required for implementation. FUDJ's integration with query optimization engines, support for native data types, and performance comparable to built-in implementations unlock new possibilities for efficient join operations. Ultimately, FUDJ facilitates more comprehensive data analysis, uncovering hidden insights, and driving accurate decision-making in diverse industries and applications.

In the subsequent chapters of this thesis, we delve into the details of both HQ-Filter and FUDJ, exploring their implementation, benefits, and experimental results. Through these innovations, we aim to revolutionize the landscape of data analysis, making it more accessible, efficient, and responsive to the evolving needs of data scientists and businesses alike.

Chapter 2

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

2.1 Introduction

The increasing availability of big data triggers a growth in the interest in data-driven applications. Many business intelligence (BI) and data exploration systems have been successfully deployed on a large scale such as Tableau, Qlik, and OmniSci. This growth resulted in a 274.3 billion dollar revenue for big data and business analytics which is more than a double increase since 2015 [44].

With the emerging adoption of data-driven applications and big data systems, the query workload significantly shifted from traditional DBMS workloads. In traditional

DBMS, the queries are usually well-crafted by users according to the schema and the DBMS can optimize these queries to return the result. On the other hand, in data exploration systems, users spend most of their time exploring, visualizing, and cleaning the data through a series of interactive queries [45]. A large number of these queries return an empty result [25,30,37,51]. There are three reasons that cause this problem. 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 result in tens, and sometimes hundreds, of queries on the database to appropriately update the visualization on the front end. Third, some of these systems generate many queries in the background to precache the result or to provide query recommendation [7,102]. In addition to the difference in query workload, data exploration systems also have different expectations of the response time. To keep the user active and engaged, the system must provide a response time of around 500 milliseconds [58]. With the large number of queries and the fixed overhead of each query, e.g., parsing and optimization, a traditional DBMS will fall short of supporting this response time. In this study, 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 [21,36,75,88] support interactivity even for very large datasets by applying several optimizations. Moreover, many systems proposed [22,29,41,57,76] 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 generated by their front-end applications.

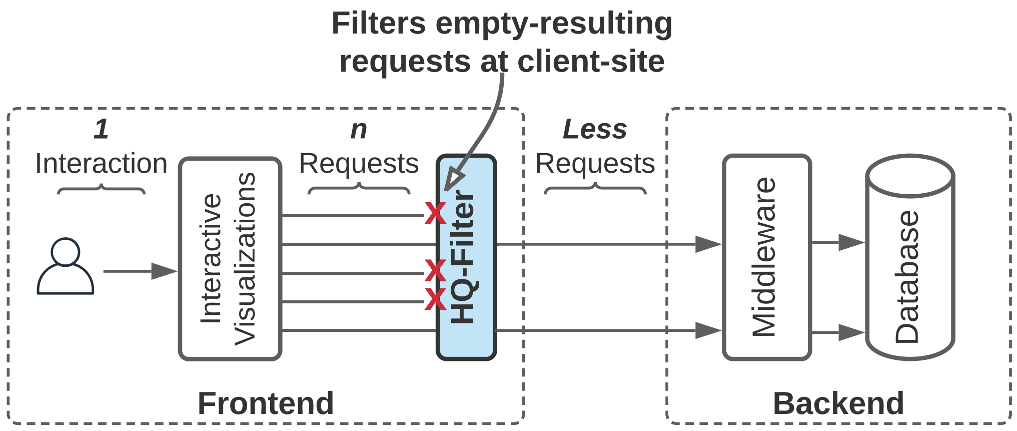


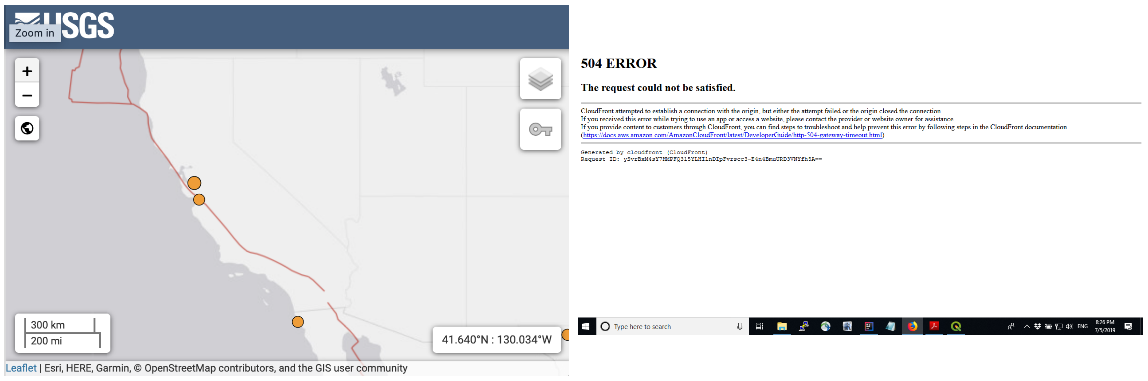
Figure 2.1: HQ-Filter in an interactive data exploration system

On the other hand, the empty-answer problem is a well studied problem which is defined as being a result of very restrictive query constraints [33]. There are techniques [25,64,66–68,91,92,96] for various domains focus on query relaxation to overcome the empty-answer problem. In query relaxation, the process starts with an initial empty-answer query, and the system either modifies the given query to avoid the empty result, recommends alternative queries, or give feedback to users what constraints causes the problem. Unlike these techniques our solution provides early pruning the empty resulting queries without sending even a single query to the backend and supports constant time detection for the multiple queries generated by the frontend, and consequently boosting the performance of all query relaxation techniques by providing an instant empty answer query to work on.

Besides query relaxation, [63] proposes a technique to store and reuse the lowest-level query parts that cause empty answers and reuse them to efficiently detect future queries that will return empty answers. Although this work complies with the notion of the iterative queries that refine the previous results of data exploration, its success is highly dependent on the previous user data and its accordance with the future ones. While this property is an advantage for traditional DBMS, for data exploration it is not since the users start without knowing what they are looking for. In our experiments, we compare this method to ours to reveal this difference. Moreover, while the time required for an empty answer check grows linearly for this approach (recall that its detection rate is proportional to the number of query parts in the storage), our method is designed to answer in constant time for any front-end generated query. Last but not least, unlike the related methods we mention here, our work can be considered as an edge computing approach as we reduce the workload on the server side by benefiting the resources of the client side.

The query workload on the server can be significantly reduced if the client can early prune empty-resulting queries without affecting the correctness of the displayed results. However, the challenge is to build a compact representation that can be quickly transferred to the client to allow the detection of empty resulting queries. For instance, a straightforward approach could identify every possible non-empty query that can be generated by the frontend, and send them to the frontend after compressing it. However, for the eBird dataset visualized in a tiled map, this filter would require 42MB of space even if we could compress it up to theoretical limits.

Motivating example 1 is an Earthquake map hosted by the US Geological Survey (USGS) [90] shown in Figure 2.2; in normal operation, most of the map tiles are empty due to the scarcity of earthquakes. However, after two major earthquakes happened in California on the 4^{th} and 5^{th} of July 2019, the system crashed due to the overload. If the system could early prune the empty tiles, the server could have survived the extra load.


Zoom in
USGS
300 km
200 mi
41.640°N : 130.034°W
Leaflet | Esri, HERE, Garmin, © OpenStreetMap contributors, and the GIS user community
Type here to search

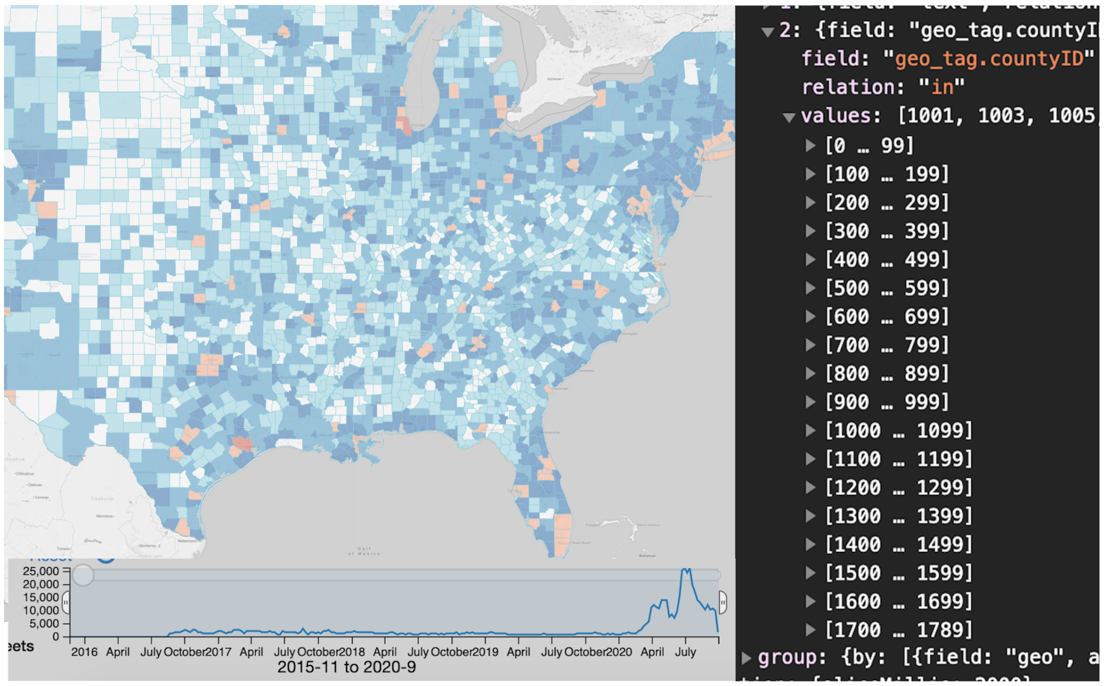
504 ERROR

The request could not be satisfied.

CloudFront attempted to establish a connection with the origin, but either the attempt failed or the origin closed the connection.
If you received this error while trying to use an app or access a website, please contact the provider or website owner for assistance.
If you provide content to customers through CloudFront, you can find steps to troubleshoot and help prevent this error by following steps in the CloudFront documentation (https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/http-504-gateway-timeout.html).

```
Generated by cloudfront (CloudFront)
Request ID: ySvrBxM4sY7HMPFQ315YLHIlnDIpFvrscc3-E4n4BmuURD3VNYfh5A==
```

Figure 2.2: USGS Earthquake Map



```
▼2: {field: "geo_tag.countyI
   field: "geo_tag.countyID"
   relation: "in"
  ▼values: [1001, 1003, 1005
    ▶[0 … 99]
    ▶[100 … 199]
    ▶[200 … 299]
    ▶[300 … 399]
    ▶[400 … 499]
    ▶[500 … 599]
    ▶[600 … 699]
    ▶[700 … 799]
    ▶[800 … 899]
    ▶[900 … 999]
    ▶[1000 … 1099]
    ▶[1100 … 1199]
    ▶[1200 … 1299]
    ▶[1300 … 1399]
    ▶[1400 … 1499]
    ▶[1500 … 1599]
    ▶[1600 … 1699]
    ▶[1700 … 1789]
▶group: {by: [{field: "geo", a
```

Figure 2.3: Cloudberry Twitter Map for Keyword “mask” with a large number of empty results

Motivating example 2 is the Twitter Map demo powered by Cloudberry [88]; users start the exploration by providing a keyword to be searched in tweets. Depending on the zoom level, Cloudberry queries the number of tweets matching the given keyword in each state, county, or city to draw the choropleth map. As shown in Figure 2.3, Cloudberry needs to query 1,800 counties simultaneously for the given region even though a significant amount of these queries will return 0 (white-colored counties). Please note that, in addition to the geographical representation, Cloudberry visualize a timeline for each day. Since "mask" is a recently popular keyword, most of the counties have 0 tweets in the previous time periods.

In both examples, the query workload on the server can be significantly reduced if the client can early prune empty-resulting queries without affecting the correctness of the displayed results. The challenge is that it is generally impractical to enumerate all empty or non-empty queries.

This study 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. The key idea in HQ-Filter is to utilize the data hierarchy to construct an approximate data structure that can filter out empty-resulting queries. Figure 2.1 shows how we utilize HQ-Filter to reduce the workload at the backend. The filter is shipped and processed at the client side which utilizes the extra power available on client machines to offload the server-side processing. In that way, the user experience can be improved and the system would handle more concurrent users without negatively affecting the data exploration tasks.

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 the network and kept in the client's main memory. In specific, we build an approximate data structure based on Bloom Filters [54] 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 the 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 [36] and Cloudberry [88]. 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, on a specific day. 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 that shows the efficiency of the HQ-Filter in improving the system performance with a negligible overhead on the client.

The contributions of this study 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.
- 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 chapter is organized as follows. Section 2.2 defines the problem. Section 2.3 describes the construction of the proposed HQ-Filter. Section 2.4 provides an experimental evaluation of the system. Section 2.5 gives an overview of the related work. Finally, Section 2.3.2 concludes this chapter.

2.2 Problem definition

We first define the terms that we use in this study by considering an interactive data exploration system that 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 spatiotemporal-textual data visualization.*

Definition 2 *(Resource ID) A unique reference is represented as a tuple that identifies a particular resource. A resource identifier for a tile in tiled map visualization can be in the form of (zoom_level,row,column), or for a spatiotemporal-textual data visualization system on Tweets can be (geo_id, keyword, start_time, end_time)*

Definition 3 *(Request) A request is a message sent from the client that contains a resource ID.*

Definition 4 *(Response) A message sent from the server to the client that contains the resource identified in a request.*

Definition 5 *(Empty Resource) A default resource that 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 tweet set for a spatiotemporal-textual data visualization system.*

Definition 6 *(Empty-Resource Identifier) If a request with a resource identifier returns an empty resource, it is called an empty-resource identifier.*

Problem Definition Consider a server that has a collection of resources R and a client sends a set of requests Q for resources by resource identifiers. Our goal is to filter the empty-resource identifiers at the client side and satisfy them without sending the requests to the server to reduce the number of requests. This can improve the response time and reduce the network traffic at both the client and server.

We further quantify how lowering the number of requests sent to a server results in a better response time in the experiments section. To give a better explanation of the problem, we give the following examples.

Example 1 - Tiled map visualization : Consider UCR-Star [36] as a data exploration system in which users are visualizing the *Parks* dataset that consists of the polygons for each park in the world. The collection of the resources R consists of more than 4^{19} (maximum zoom level is 19) map tiles at the server. Any tile request, specified by a zoom level, row, and column number, is a query sent from the client to the server. If the resulting tile of a query q is an empty tile, i.e., none of the parks intersects the tile, we define q as an empty resulting query. Hence, the problem is how a client determines that a tile is empty without contacting the server.

Example 2 - Spatiotemporal-Textual data visaulization: Consider the Twitter Map powered by Cloudberry [88] as an interactive data exploration system in which users enter a keyword and the system visualizes a choropleth map with the number of tweets in each state, county, or city, depending on the zoom level. The system also visualizes a timeline histogram with the number of tweets per day. For this specific example, our collection of resources R consists of the total number of tweets for each keyword k grouped by each level of details and for each day. The resource ID is the combination of a keyword k, a level-of-detail ID l, e.g., city ID, or county ID, and a date d. A resource is empty if the count is zero. Therefore, the problem is determining if the result of the query $q = (k, l, d)$, will be zero or not before sending q to the server.

2.2.1 Baseline 1: Exact Approach

A possible solution to this problem is to ship the list of all non-empty resource IDs to the client to filter requests. However, this approach is not practical due to the large number of resources. For example, the eBird dataset hosted at UCR-Star contains 24 million resources which would take about 180 MB of storage and more than a minute to transfer on a high-speed connection.

Alternatively, we can represent the empty and non-empty resources as a bit sequence where each resource is represented by a single bit that is set if the resource is non-empty. Then, we can compress the bit sequence to minimize its size. However, this would still be too large to be practical.

To find the theoretically optimal compression limit for the bit map, we use Shannon's Entropy formula, where p_i is the probability that a bit is equal to i, as below.

$$H(X) = -(p_0 \log_2 p_0 + p_1 \log_2 p_1)$$

For the same UCR-Star dataset mentioned above, this would take about 42 MB which is still impractical.

2.2.2 Baseline 2: Probabilistic Approach

Since the exact approach is not practical, this leaves us with probabilistic approaches. Being one of the most popular probabilistic data structures, we consider *Bloom Filters (BF)* [54] as a baseline solution in this study. A BF represents a set of resource IDs and supports a membership query that, given a resource ID, can answer either *may be in the set* or *definitely not in the set*, i.e., false positives are possible. The first question we should ask is whether to insert *empty* or *non-empty* resource IDs. If we insert empty resource IDs, then a false positive would identify a non-empty resource as empty which will result in an incorrect behavior so we disqualify this option. On the other hand, if we insert non-empty resource IDs, then a false positive would identify a non-empty resource as empty. This will result in a request sent from the client to the server for an empty resource and the result will be empty. Even if it results in an unnecessary query, the system will still behave as expected so we decide to insert non-empty resources in BF. Additionally, we have to insert *all* non-empty resource IDs otherwise a negative result from BF cannot be trusted to be an empty resource.

To evaluate the expected performance of BF, let us provide examples for eBird and Cemetery with approximately 24 million and 5 million non-empty resources, respectively. Since a user can go up to level 19, we need to add all of the non-empty resource IDs up to this level to handle emptiness checks for all possible requests. Also, we assume that our memory budget is 512KB. Now, we can calculate the optimum number of hash functions k and the false positive probability p for each dataset which yield $k = 1$ and $p = 0.727$ for Cemetery, and $k = 1$ and $p = 1$ for eBird datasets. As a result, we can state that 27.3% of the requests for empty resources can be filtered potentially (performance of the filter) for the Cemetery dataset while it is 0 for the eBird dataset with the 512KB filter size. If we could enlarge the filter size to 3MB, performances of the filter could be 90% for Cemetery and 39% for the eBird datasets.

2.3 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 approximate filter that stores only a subset of the non-empty resource IDs and utilizes the data hierarchy to check for any resource ID. To further explain the key idea we present a toy example in Figure 2.4 which consists of 21 tiles organized in 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.

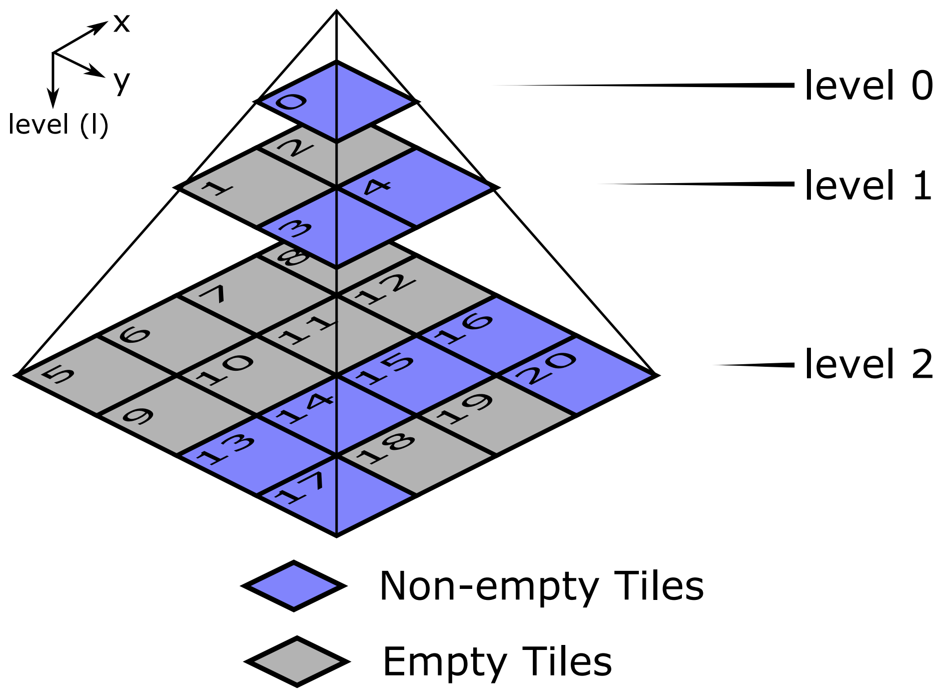


Figure 2.4: Three levels of tiled map visualization shows empty and non-empty tiles

First, if we build a Bloom Filter (BF1) with the 9 non-empty tiles, the theoretically optimal false positive rate for BF1 is $p_1 = 0.675$ 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 $p_2 = 0.278$. If a query asks for one of the tiles in levels 0 and 1, it can be answered 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 for sure 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 could 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 level at which BF2 should be constructed to achieve an overall optimal performance. Furthermore, if the data has multiple hierarchies, then we need to find the optimal level for each of these hierarchies that together reach the optimal performance, e.g., space and time hierarchies. In this study, 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.

2.3.1 Performance Metric

In this part, we make a formal definition to quantify the performance of a constructed filter. Our key idea is to utilize the inherent hierarchy in the data. For example, in tile-based map visualization, all tiles are organized in a hierarchical quad-tree-like pyramid structure with one root tile at the top, and each tile has four children as shown in Figure 2.4. In spatiotemporal-textual data visualization, there are two hierarchies that can be defined, a geospatial administrative hierarchy, i.e., country, state, city, and ZIP code; and date-time hierarchy, i.e., year, month, and day. In both cases, we can organize the resources, i.e., tiles or tweet counts, into a hierarchical structure. The key observation is that if a resource is empty, then all the resources below it in the hierarchy must also be empty. Below, we make some definitions that we use in this chapter.

Definition 7 *(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 8 *(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 9 *(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 10 *(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 the following. If a resource is empty, then all its descendants are also empty. This means 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. Next, we define the performance of a filter P as the *expected* number of resources that it can correctly detect as empty resources. The higher this value, the more requests it can filter at the client side. This metric is formally defined below.

$$P(l_f) = (1 - p_f) \times \left(\sum_{l=0}^{l_f} |E_l| + \sum_{r \in E_{l_f}} g(r, l_{max}) \right) \tag{2.1}$$

Where,

- P is the performance of the filter measured as the expected number of empty resources that the filter can detect,
- p_f is the false positive probability of the HQ-Filter f,
- l represents a level in the hierarchy such that $l \in 0, 1, 2, \ldots, l_{max}$,
- l_f is the level of the HQ-Filter f,
- E_l is the set of the empty resource identifiers (E-RIDs) at level l
- l_{max} is the maximum level of resources that the client can request,
- $g(r, l_{max})$ is the 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 level $0 \leq l \leq l_f$. 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 Equation **??**.

The first summation in the equation above 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.

Definition 11 *(HQ-Filter Construction Problem) Given the set of empty resource identifiers E_l for each level l and a space constraint m, find the level l_f of the HQ-Filter that maximizes the performance P as defined in Equation 2.1 and construct that filter.*

2.3.2 HQ-Filter Construction

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., $\langle |N_0|, |N_1|, \dots, |N_{l_{max}}| \rangle$. 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 2.1 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. 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 count NE for all the levels due to the excessive number of non-empty resources at the deep level. In this case, we propose an optimized counting technique that counts only the levels that are needed for the next step. We show how to implement this step efficiently on both Spark and AsterixDB.

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 2.1 for each level and choosing the best. The main challenge is to efficiently compute the bounded cardinality of each record, $g(r, l_{max})$, as described in Definition 10.

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 level 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.

To wrap up, the three steps to construct the HQ-Filter are summarized here. 1) For each record in the input, find the matching RIDs in each level and count them by level. 2) Given a memory budget m, the count of non-empty RIDs per level, and the maximum request level l_{max}, find the optimum level l_f which maximizes the performance P. 3) Given the optimum level l_f, construct the HQ-Filter by inserting all non-empty resource IDs in levels $[0, l_f]$.

2.3.3 HQ-Filter for tile-based Map Visualization

In this section, we give the concrete algorithm that constructs HQ-Filter for the case of tiled map visualization in UCR-Star [36]. To make a concrete algorithm, we have to do four 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. Finally, we show how to construct the filter once the level l_f is found.

Tile Hierarchy: In the tiled map visualization, the resources comprise tiles organized in a quad-tree-like pyramid 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 $(l - 1, \lceil x/2 \rceil, \lceil y/2 \rceil$. 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. First, we use the `mapPartition` Spark transformation to run a local step on each 128 megabyte partition that builds a set of non-empty resource IDs per level N_l, where $0 \leq l_{max}$. In this step, each input record is first mapped to a tile at the deepest level l_{max} which is a constant-time operation. That tile is added to the set of tile IDs $N_{l_{max}}$. Then, we recursively find the parent tile and repeat until the root tile is reached. All these tile IDs N_l are stored as hash sets to ensure a constant-time process per insertion. 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. Finally, we produce the final result as the size of each final hash set. This algorithm is highly-parallel since each step runs in parallel. However, it has a bottleneck in merging the deepest level l_{max} which contains the largest number of non-empty tiles. We show in Section 2.3.5 how to further improve this algorithm.

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 use case.

Due to the uniformity of the tile structure, where each tile has exactly four children, computing the bounded resource cardinality can be computed efficiently as follows.

$$\begin{aligned} g(r, l_{max}) &= \sum_{l=l_r+1}^{l_{max}} (4^{(l_{max}-l)+1}) \\ &= \frac{4 \times (4^{l_{max}-l_r} - 1)}{3} \end{aligned} \tag{2.2}$$

As a result, when we apply the $g(\cdot, \cdot)$ for multi-level visualization to Equation 2.1, we can simplify the equation as follows.

$$P(l_f) = (1 - p_f) \times \left(\sum_{l=0}^{l_f} |E_l| + |E_{l_f}| \frac{4 \times (4^{l_{max}-l_f} - 1)}{3} \right) \tag{2.3}$$

Equation 2.3 is much more efficient than Equation 2.1 because it does not have to iterate over each empty tile at level l_f. This is only possible because all tiles at level l_f have the same exact bounded cardinality which means we can just multiple that cardinality by the number of empty resources $|E_{l_f}|$.

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 2.3 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 a 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 [] and hashing method from [54], we hash the unique Tile IDs and set k number of bits to 1 in the bit array.

Client-side Implementation: UCRStar's front-end is built as a web application using OpenLayers. It creates a tiled map layer that requests tiles with their IDs in the form (l, x, y). To integrate the HQ-Filter, 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. If it is in the filter, we request it, otherwise, we skip the call. If the level $l > l_f$, we find the ascendant 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. Then, we test if the ascendant tile is in the HQ-Filter in the same way described above to decide if we should skip the tile. This test runs in constant time which adds a negligible overhead on the client.

2.3.4 HQ-Filter for Spatiotemporal-Textual Data Visualization

Similar to the previous section, we focus on four parts. Based on how Cloudberry [88] 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 an 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, $\ldots$ 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 $0 \leq 0 < (l_{geo-max} + 1)(l_{time-max} + 1)$. In this case, we can easily convert back and forth using the following equations:

$$l = (l_{geo-max} + 1) \cdot l_{time} + l_{geo} \tag{2.4}$$

$$l_{time} = \lfloor l/(l_{geo-max} + 1) \rfloor \tag{2.5}$$

$$l_{geo} = l \mod (l_{geo-max} + 1) \tag{2.6}$$

The linearization of the levels from multiple hierarchies makes it easier to apply Equation 2.1 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 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 an 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 can define the bounded resource cardinality 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 2.1. 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 an 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 [88]'s front-end is implemented as a web application similar to the UCR-Star [36]. 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 each for a specific geolocation and date. The request level (l_{geo}, l_{time}) is compared to the HQ-Filter level l_f. If both are less than or equal to their corresponding filter level l_f, we directly test if the resource is non-empty in the filter. Otherwise, we need to 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.

2.3.5 Optimizations for Counting

For both case studies described earlier, the first step, 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 at deep levels. For example, in the `Parks` dataset which has only 10 million polygons, the number of non-empty resources in levels 17, 18, and 19, is 720M, 2.8B, and 11B, respectively.

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 2.1, 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.

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. The algorithm we showed earlier finds the exact count at all levels which might not be necessary. This part proposes an alternative algorithm that approximately counts the number of resources per level. We

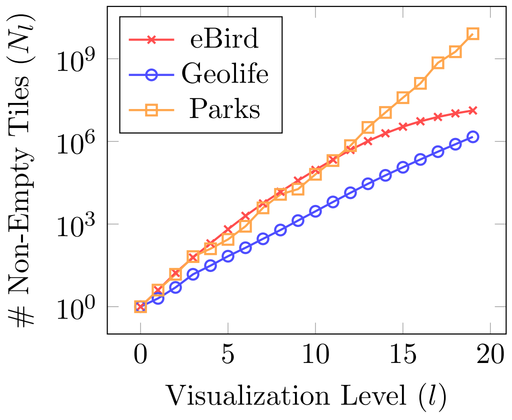


Figure 2.5: Tiled Map Datasets

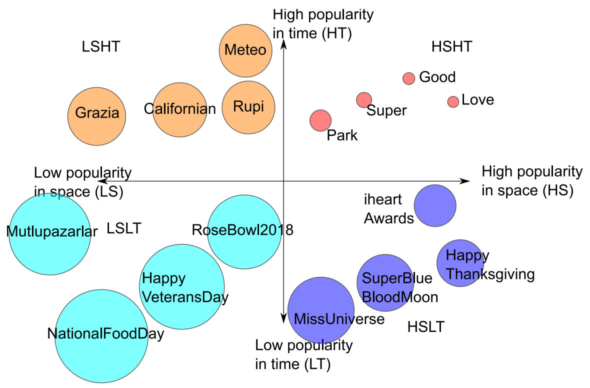


Figure 2.6: Classification of keywords according to popularity in space and time

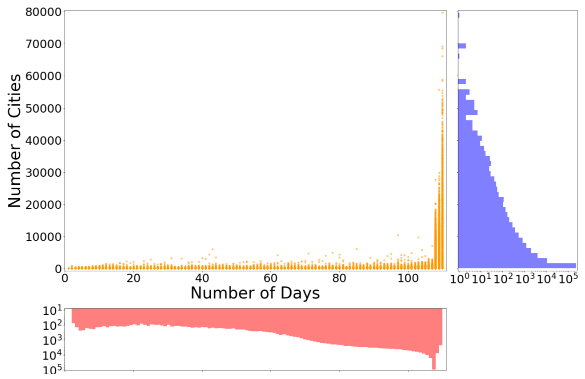


Figure 2.7: Spatial and temporal distributions of keywords

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 [42] 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 2.3.3. 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.

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. In other words, it keeps going deeper into the levels as long as the performance increases. Once the performance starts to decrease, the algorithm stops and returns to 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.

2.4 Experiments

We evaluate the HQ-Filter applied in two real-world applications: UCRStar [36] and CloudBerry [88].

2.4.1 Setup

Software and Hardware setup

We compare HQ-Filter with two baselines: **No-filter** when all the requests are sent to the server, and Bloom Filter (BF) when all non-empty resources are added to a regular BF. For the two case studies, we have two implementations: one in Spark for tiled map visualization, and the other in AsterixDB for spatiotemporal-textual visualization. All experiments run on an Amazon Web Services (AWS) cluster with 20 machines of type **m5.xlarge** which has 4 vCores, 16 GB memory, and 200 GB SSD. The single-machine server resides in the North California data center while the client locations are varied between Oregon, North Virginia, and Frankfurt.

Datasets and Workload

Datasets: Table 2.1 lists the datasets that we use. The first three are used with tiled map visualization and are available at UCR-Star [36]. The fourth is used with spatiotemporal-textual data exploration. Figure 2.5 shows the number of non-empty tiles per level for the datasets used with tiled map visualization which highlights the characteristics of these datasets. Most notably, the Parks dataset has about an order of magnitude more non-empty tiles in deeper levels. For the Twitter dataset, Figure 2.7 shows a scatter

Table 2.1: HQ-Filter Evaluation 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

plot where each point represents a keyword, and the x and y axes show the number of days and cities in which this keyword appears, respectively. For example, a word in the top-right corner appears on most days and in most cities. Based on that figure, we classify the words along the two dimensions into high and low popularity. If a keyword has less than the average along each dimension it is marked with low popularity, otherwise, it has high popularity. Thus, we end up with two classifications low/high popularity in time (LT/HT), and similarly, low/high popularity in space (LS/HS). Considering all the combinations, we end up with four categories of keywords. Figure 2.6 shows examples of keywords in 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. For example, words like ‘love’ and ‘good’ are in HSTS, hence there are only a very few empty resources for them. On the other extreme, a word like ‘NationalFoodDay’ is categorized as LSLT and hence has a lot of empty resources that the HQ-Filter can potentially save.

Workload: Based on each dataset and 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 provides the ability to control the ratio of dense and sparse users.

Parameters & Metrics

Parameters In our experiments, we vary the filter size (m), dataset size, ratio of dense users, and number of users.

Metrics In our experiments we measure the HQ-Filter creation time, average response time per client request, server workload, and the number of eliminated empty queries.

2.4.2 Accuracy of the Performance Metric (P)

This experiment shows the accuracy of the proposed performance metric P in representing the system performance. Thus, we can use it as the main performance metric in constructing and evaluating the HQ-Filter. Figure 2.8 compares P (represented as dashed

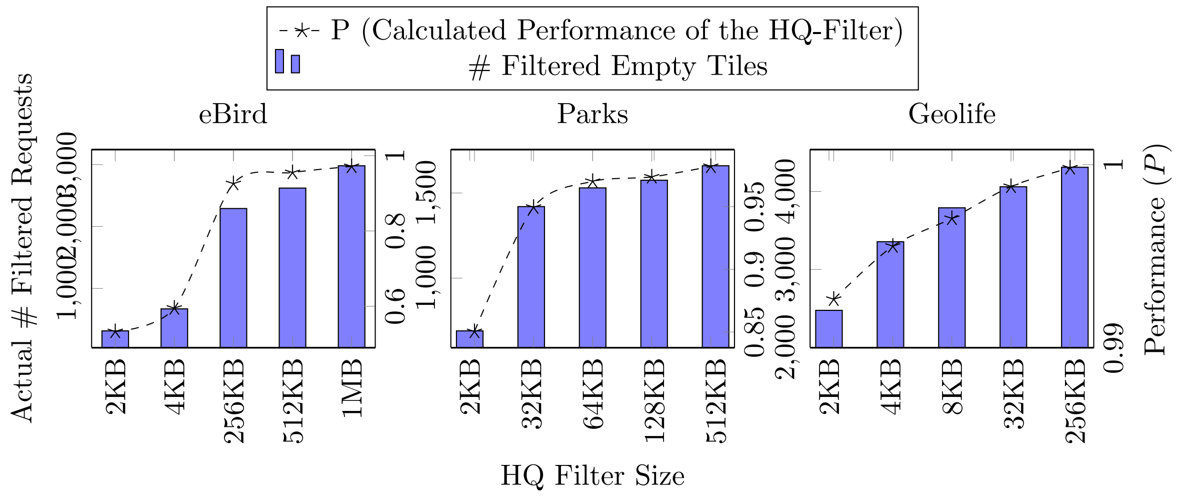


Figure 2.8: Actual number of filtered requests for various size HQ-Filters along with their computed P metrics

lined) to the actual number of filtered requests (represented as bars). In this experiment, there is an equal mixture of dense and sparse users in the tiled map visualization study case. We vary the filter size from 2KB to 1MB and report and show the sizes that make the biggest difference to highlight the relationship between the two metrics. The coherence between P and the actual number of filtered empty resulting queries can be clearly observed in Figure 2.8. It is worth mentioning that, for Geolife, although P varies in a small range (0.99-1.0), it still shows a strong correlation with the actual number of empty requests. The reason for this tight range is that it is a highly skewed dataset with a small number of non-empty resource ids. Notice that the estimated performance P is calculated based on the total number of empty resources which makes it very high for deep levels that usually contain a huge number of empty resources. However, the actual performance is based

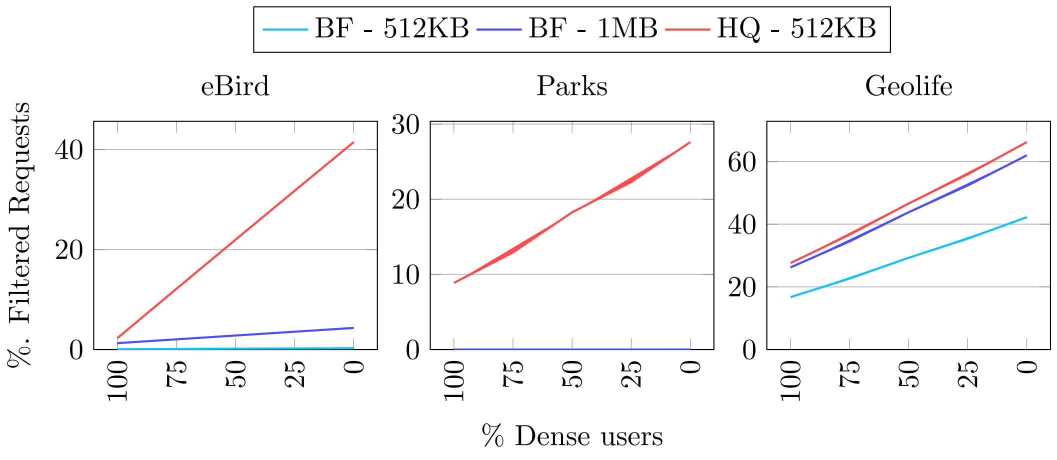


Figure 2.9: User Types vs Filters

on specific user behavior that does not request all the records. However, the estimated performance P is still good in estimating the potential saving.

2.4.3 Effect of User Types

This experiment studies the effect of the types of users on the performance of the filter. Recall that *sparse* (*dense*) users who are interested in areas with a large (small) number of empty resources. We vary the ratio of dense users while keeping the total number of users fixed.

Tiled map visualization: Figure 2.9 compares HQ-Filter of size 512KB with two Bloom Filters (BF) of sizes 512KB and 1MB. 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 can make the following observations from the results. First, HQ-Filter is consistently better even when it occupies half the size of BF. This is a direct result of accounting for the data hierarchy which allows

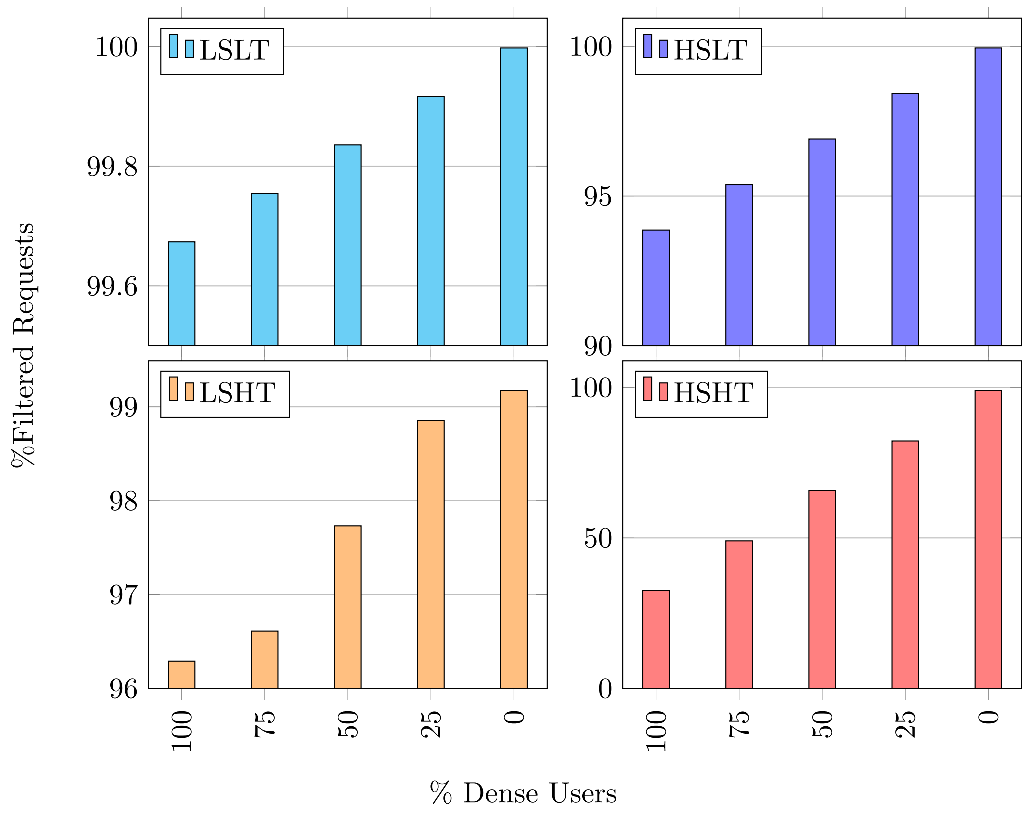


Figure 2.10: Filtered requests for various user combinations for all keyword categories

it to detect more empty resources with less size. Second, the fewer dense users, the better the filters behave since there is a bigger pool of empty requests to filter. Third, for the Parks dataset, both BF versions behave poorly due to a large number of non-empty resources (as shown in Figure 2.5). On the other extreme, BF performs the best with the Geolife dataset that has the least number of non-empty resources but HQ-Filter is still better while using less memory. Fourth, this experiment reveals that BF is not competent to HQ-Filter due to its false-positive rate approaching 1.0 even with a reasonable 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 the network to the client.

Spatiotemporal-Textual Data Exploration: Similar to the previous experiment, we test the HQ-Filter when integrated into the Cloudberry application. In Figure 2.10, we vary the percentage of dense users and measure the percentage of filtered requests. We repeat the experiment for the four classes of keywords as defined earlier in this section. 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 we assume each user zooms into a different country. As a result of that and the language differences among those countries, our sparse users for each keyword category have almost all of the requests for empty resources. In addition, we also observe that the category of the keyword affects the performance where the highly popular keywords in space and time (HSHT) have the least performance, especially, with more dense users due to a small number of empty resources that the filters can detect. On the other extreme, the keywords with low popularity (LSLT) yield the highest performance even when dense users use the system. The other two categories which have high popularity in one dimension only (either space or time), fall in between these two extremes. Finally, the performance of this experiment is much higher than the tiled map visualization since there is a higher number of empty resources due to the keyword filter which reduces the number of non-empty resources.

2.4.4 Response Time Improvements

It is crucial to show that eliminating the requests for empty resources improves system performance. Figure 2.11 shows the effect of eliminating empty-resulting queries on both the client and server performance. In this experiment, we vary the number of filtered

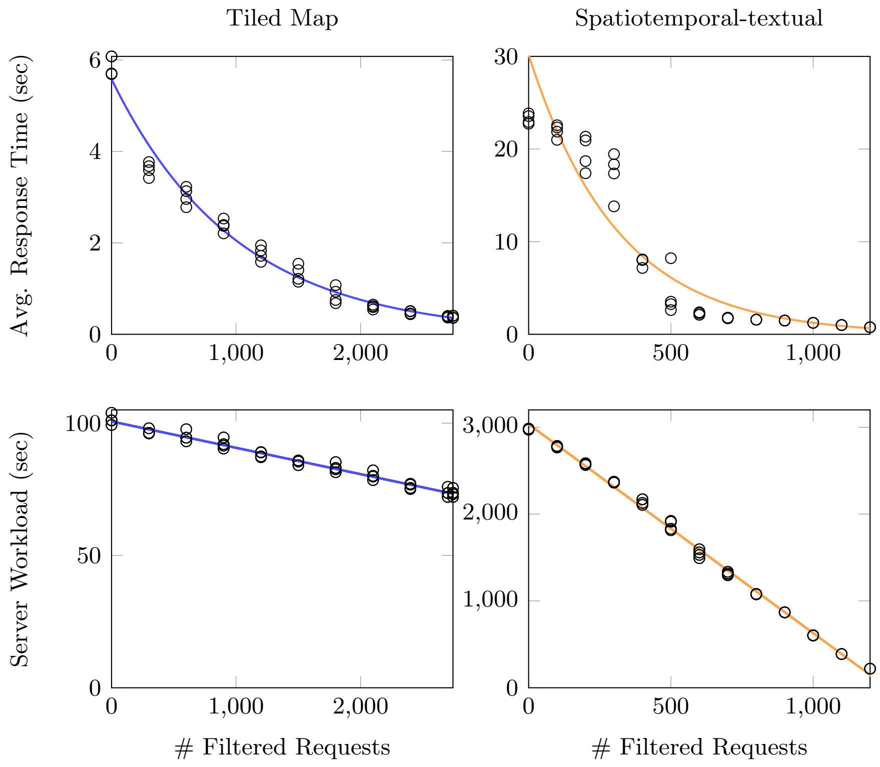


Figure 2.11: The effect of filtering empty requests on the client response time and server workload

requests on the x-axis and measure the performance. We vary this number in a controlled way to accurately measure its effect and we repeat each point four times to ensure the accuracy of the result. We use the eBird dataset and the 'love' keyword for the tiled map and spatiotemporal-textual exploration applications, respectively. The workload consists of eight dense and eight sparse concurrent users.

Average Response Times (client-side): The top two figures show that as we filter more empty resources, the average response time observed on the client side 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

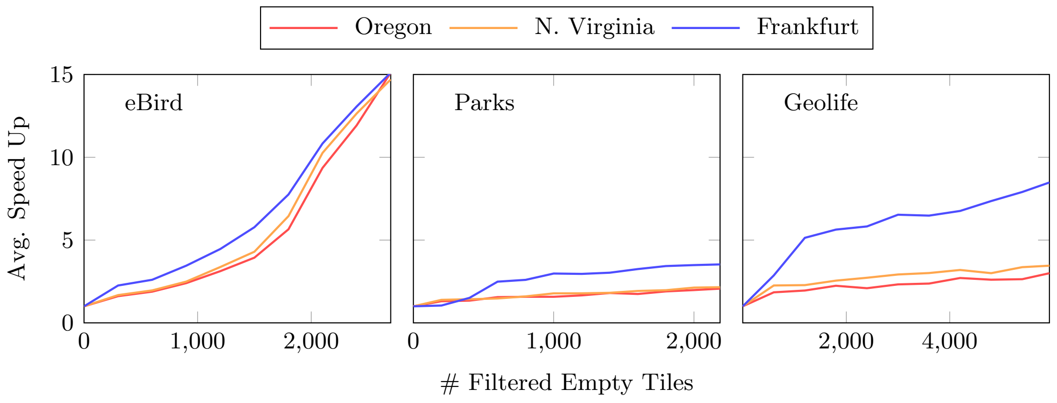


Figure 2.12: Average speedups for three different client locations

data exploration, respectively. This is a result of saving the transmission of the request over the network and the processing on the server side. It is worth mentioning here that the times converge to a fixed value which represents the time needed to process the non-empty requests since the filter only saves the empty requests.

Server Workload: The two figures at the bottom of Figure 2.11 shows 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 are typically sent by its front-end design.

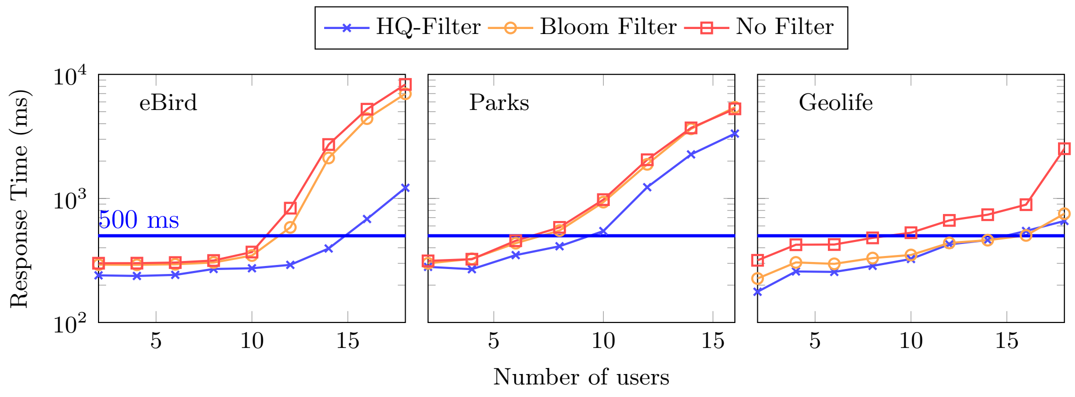


Figure 2.13: Server performance with the usage of HQ Filter (512KB), Bloom Filter (1MB), and NO Filter (without any filtering)

To further study the effect of filtering requests, we measure the speedup observed at the client side but this time we run the client at three different locations, Oregon, N. Virginia, and Frankfurt, while fixing the server in N. California. All clients are AWS machines of the same instance type. Figure 2.12 shows the result of the tiled map visualization application with the three datasets. The results show that as the client gets further away from the server, the speedup observed on the client side increases. This is due to saving the request network roundtrip time which gets bigger as the client gets further away.

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 since they do not depend on the system load or hardware specification.

2.4.5 Improvement on server capacity

This experiment studies how the use of HQ-Filter can increase the server capacity in terms of the number of concurrent users that can be supported while providing a processing time of less than 500 milliseconds. Figure 2.13 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.

2.4.6 HQ-Filter Creation

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

Tiled Map Visualization

Counting: The most expensive step of HQ-Filter creation is the counting step. As Figure 2.5 shows, the number of non-empty resources can sometimes increase excessively for deep levels which makes the counting step very expensive. We proposed three counting techniques, exact (E), approximate (A), and incremental (I). Figure 2.14 shows the total time for the counting step of the three techniques as we increase the maximum level of the filter l_{max} from 10 to 19. For Geolife, as a result of having a small number of non-empty tiles, any of our counting techniques perform similarly for any visualization level. However, the best one is the exact counting because being a lean approach does not include optimizations which requires additional work.

For eBird, since the number of non-empty tiles in deeper levels is larger, we start to see a difference between the three techniques. The running time for the incremental technique stabilizes after 12 levels since this is the optimal level for this dataset, and the incremental technique always terminates after finding the optimal level regardless of the maximum level. The exact technique is still faster since it is simpler and more direct.

Finally, 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 at 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.

Breakdown of Creation Time: Figure 2.15 shows the breakdown of the overall construction time for the three counting methods. Since the incremental method combines the counting and optimization steps, we report these two steps together for all three methods but the time of the optimization step is negligible. To avoid excessive running time, we set

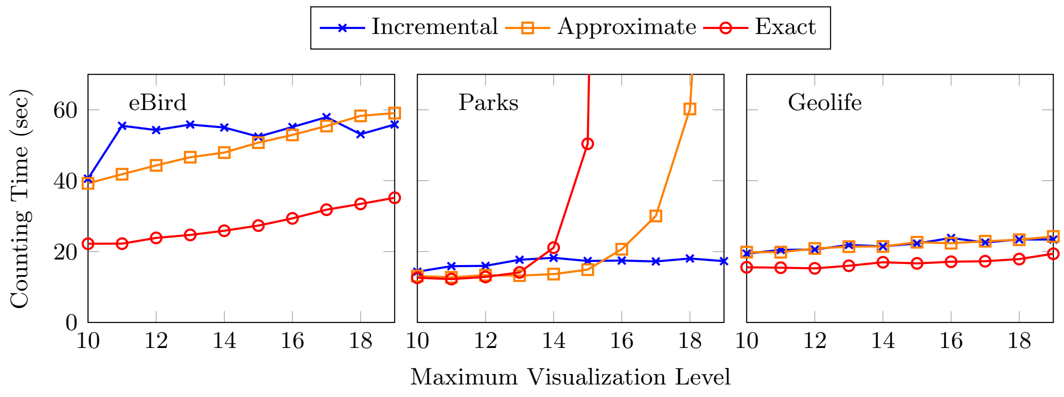


Figure 2.14: Non-empty tiles counting times for levels 10 to 19

the maximum level to 15. As noted earlier, the exact count is generally faster except for the Parks dataset due to the large number of non-empty tiles. This experiment reveals an interesting finding for the construction step. For both approximate and exact methods, the construction time is the same since they first count all the levels and then construct the filter for the chosen level. However, for the incremental method, the construction time is much smaller since we can cache the tile IDs for the last counted level which can then be used directly to construct the filter without scanning the dataset again.

Spatiotemporal-Textual Data Visualisation

Figure 2.16 shows the breakdown of the HQ-Filter construction process in the spatiotemporal-textual data exploration application. Following the design of Cloudberry [88], 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

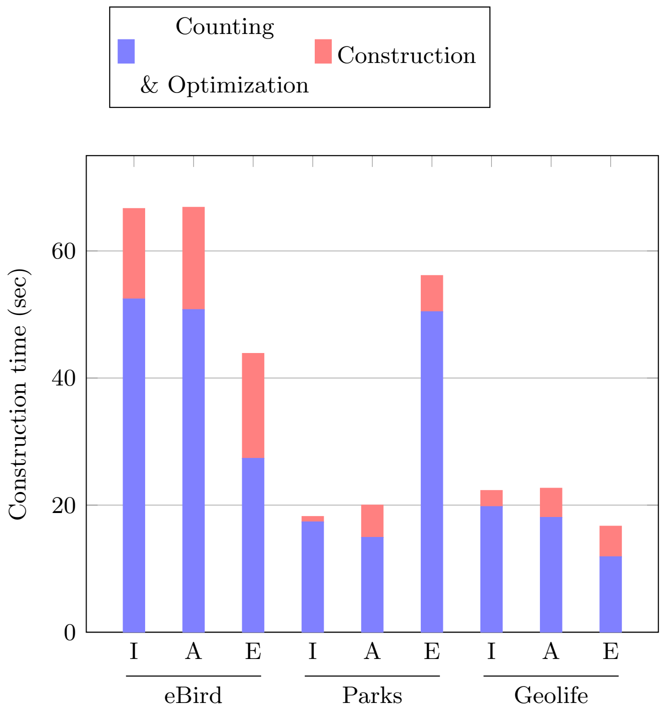


Figure 2.15: Tiled map with Incremental (I), Approximate (A), and Exact (E) counting methods

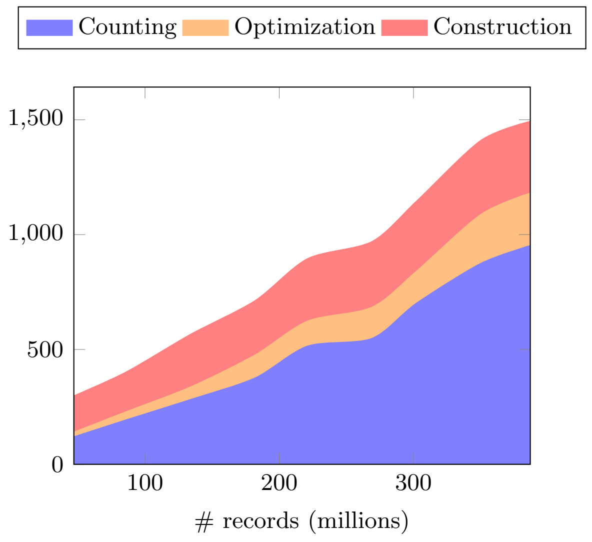


Figure 2.16: For spatiotemporal-textual data application in AsterixDB

expensive step. Since we keep the resource ids in memory after counting, the construction step remains stable as compared to the counting step. The optimization step for this application is slightly more expensive 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.

2.5 Related Work

Existing work in the area of data exploration can be generally classified based on the layer they focus on in three categories, presentation, application, or data layer [45]. Notice that some work belongs to more than one category if they contribute to more than one layer.

2.5.1 Presentation Layer

Work in this category focuses on providing an interactive interface that allows users to explore data. Vizdom [21] builds a visualization system that supports user interaction with pen and touch. Couldberry [88] provides a front-end for searching and visualizing tweets on a map by aggregating the results at the city, county, or state level. Polaris [87] (commercialized as Tableau) visualizes the queries with a set of charts based on the *grammar of graphics* algebra [100].

D3 [11] is a highly popular JavaScript data visualization library that supports a vast variety of data visualizations and interactivity features.

Above mentioned works helped in building many interactive exploration systems that encourage more users to explore the data. Consequently, this resulted in an increase in the number of queries including the ones that generate an empty result. The proposed work can be effectively used to improve the performance of all these systems by removing the extra overhead caused by processing the empty resulting queries.

2.5.2 Application Layer

The work in this category 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. Cloudberry [88] proposes a middleware component in which frequent queries are materialized and a mechanism implemented to break a complex user request into smaller queries to provide progressive results [48]. In this study, we describe how to integrate HQ-Filter into Cloudberry to improve its performance by skipping empty-resulting queries.

Some applications follow the *Approximate Query Processing (AQP)* approach [55, 74, 83] in which they use sampling to speed up the query processing of big data. The goal is to maintain the interactivity of the system at the cost of an approximate result. While HQ-Filter is also an approximate technique, it has a different goal than the work in this category. The goal of the proposed work is to skip empty resulting queries while regular queries are still processed using regular techniques. Even though HQ-Filter is an approximate technique, the final result is still exact since it only skips queries that are guaranteed to be empty. In summary, HQ-Filter is orthogonal to the work listed above which also means that they can be combined together where the HQ-Filter can skip empty resulting queries and these techniques can answer non-empty resulting queries.

Query relaxation is a well-studied topic which is related to the empty-answer problem [33]. Given a query that returns an empty answer, the query relaxation techniques, e.g., [66, 68, 92], 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.

2.5.3 Database layer

The work in this category focuses on storing and indexing the data to serve interactive exploratory systems. The Data Visualization Management System (DVMS) [105] proposes a database system that pushes the visualization query awareness into the core of the database. Tableau [99] uses the VizQL [87] visualization query language to answer the visualization queries efficiently. HadoopViz [32] and GeoSparkViz [110] use a distributed query engine for visualizing big spatial data using the tiled map approach. AID* [34, 35] proposes an adaptive index that minimizes index size and construction time for tiled map visualization. NanoCubes [56] builds an in-memory multidimensional cube to visualize spatiotemporal aggregate heatmaps. imMens [59] uses an aggregation-based method for big data visualization with a front-end that uses WebGL. VAS [71] and Tabula [109] use sampling to downsize the data while reducing the error in the produced visualization. GloBiMaps [98] 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.

HQ-Filter can be constructed efficiently at the database layer and then shipped to the application or presentation layers to reduce the number of queries that the database layer has to answer. Therefore, if a large number of empty-resulting queries is expected, HQ-Filter can be combined with any of the above approaches to reduce the workload on the database layer.

2.6 Conclusions

HQ-Filter is a filter for queries that return the empty result in visual data exploration systems. As a trial-error process, data exploration tends to deal with a large number of this type of queries. HQ-Filter takes advantage of the natural hierarchy of the data and provides filtering for all visualization levels in the user interface, which is a challenging problem due to the high number of query possibilities. We address the challenges of this problem and provide an efficient algorithm to implement HQ-Filter for tiled map visualization and spatiotemporal-textual data visualization systems. Our experiments show that for even large datasets, HQ-Filter can be created at reasonable times and it improves the performance of the systems by increasing the capacity of concurrent users that are handled within the interactivity limits, average response time for each user, and reduces the server workload.

Chapter 3

FUDJ: Flexible User-defined Distributed Joins

3.1 Introduction

Joining datasets is a fundamental task in data analysis that has been extensively studied for decades [77]. Historically, Database Management Systems (DBMS) treated ”join” as an operation for structured data with simple conditions like equality. However, with the growing volume and diversity of data, along with the rise of data-driven applications, various other types of join operations are becoming increasingly popular. Today, data scientists often need to combine large and diverse datasets from sources like social networks and IoT devices using distributed systems. This calls for optimization techniques to enhance join query performance, especially as these new join types involve complex conditions and data types from diverse and less structured sources. As a result, there has been a significant amount of research work in the area. However, the availability of optimization techniques for the new join types in DBMSs still remains limited due to implementation and integration complexities as explained below.

Currently, there are three methods for implementing new join operators. First is the **standalone** [12, 19, 28, 43, 50, 89] approach, where developers independently craft algorithms without any platform integration. Second is the use of the programming paradigm of a **distributed system**, such as Spark [82, 111], Hadoop [1, 26, 27, 93], or Flink [49, 80]. These two approaches often yield tailored applications for specific query types, resulting in inflexible and non-adaptable systems. Hence, they struggle to optimize join queries that involve diverse and complex conditions since they are not integrated into a holistic query optimization engine like in DBMSs.

Besides these methods, a few studies have proposed implementing join optimization techniques as **built-in** operators within a DBMS context. For instance [17, 52] study set-similarity and interval joins on AsterixDB, [72] studies spatial join on Paradise, and [84] studies set-similarity join on PostgreSQL. These approaches demonstrate that incorporating new join algorithms in DBMSs has clear benefits such as seamlessly integrating optimized joins with other optimizations and enabling result pipelining for further processing. However, they do not offer a universal implementation model for other join types to follow. Consequently, each new join method still requires implementation from scratch, and the availability of DBMSs capable of accommodating an array of optimization techniques is limited. This gap often leads to complex join queries being processed through **on-top** solutions, wherein a User-defined Function (UDF) implements the predicate function and a

Nested Loop Join (NLJ) operator handles query processing. Unfortunately, this results in slow query execution times that can affect the pace of the data analysis process.

3.1.1 Motivation

To better clarify the importance of complex join query optimization, we provide an example scenario with three queries. Consider a data science team that wants to identify which parks were affected by wildfires in the last year by using the "Wildfires" and "Parks" datasets with the schemas shown below:

```
CREATE TYPE Parks_Type {id:uuid, boundary: geometry, tags: string}

CREATE TYPE Wildfire_Type {id:uuid, lat: float, lon: float, fire_start: datetime,

    fire_end: datetime}
```

Type 3.1: Parks and Wildfires Type Definitions

```
SELECT p.id, p.tags, p.boundary, COUNT(w.id) AS number_of_fires

FROM Parks p, Wildfires w

WHERE ST_Contains(p.boundary, ST_MakePoint(w.lat, w.lon))

    AND w.fire_start >= parse_date("01/01/2022" , "M/D/Y")

GROUP BY p.id, p.tags, p.boundary

ORDER BY number_of_fires DESC;
```

Query 3.1: Spatial Join Query

To have a list of recently damaged parks, the data science team wants to run the spatial join query below with the computationally expensive predicate $STCONTAINS$ that detects whether one spatial object (wildfire locations as points) is contained by another one (park boundaries as polygons). Note that the Query 3.1 is not only a join query but involves other operations like filtering, aggregation, and sorting.

Even though the spatial join problem has been well-studied and there are several papers proposing successful approaches, it is rare to find a general-purpose distributed big data processing system that provides an efficient query execution plan for Query 3.1 out of the box. Since the join condition is not equality and the data type is geometry, fast and well-know solutions such as Hash Join (HJ) cannot be utilized and the NLJ operator has to be used which results in prolonged running times. One could argue that there are several extensions, packages, or systems available for spatial operations. However, for ad hoc queries they are not a satisfactory answer since efficiently integrating them is not a simple task.

After Query 3.1, a member of the team may want to find alternative parks for the ones that are damaged by the wildfires to recommend to potential visitors. This might be done with Query 3.2 by listing parks that have similar "tags" for each damaged park since tags are used to describe the properties of the parks with words like "River, Scenic Landscape, Camping, Backpacking".

```sql
SELECT dp.park_id, p.id, jaccard_similarity(dp.tags, p.tags) as similarity
FROM Damaged_Parks dp, Parks p
WHERE dp.park_id <> p.id
   AND jaccard_similarity(dp.tags, p.tags) >= 0.5
ORDER BY dp.park_id, similarity;
```

Query 3.2: Text-similarity Join

Next, another member of the team may want to investigate the relationship between the weather and wildfires by using the "Weather_History" dataset with the schema defined in Query 3.2. Assume that damaged parks are stored in the "Damaged_Parks" dataset.

```
CREATE TYPE Weather_History_Type {id:uuid, location: geometry, reading_interval:
    interval, temp: int}
```

Type 3.2: Weather_History Type Definition

To find the average temperature for each wildfire that has happened in each park, they can use Query 3.3 which is a combination of spatial and interval joins. Query 3.3 finds the weather readings close to the wildfires that happened in each park by using predicates *ST_DISTANCE* and *ST_CONTAINS*. Then, by using *overlapping_intervals*, it detects whether two intervals, weather sensor reading intervals and wildfires, are overlapping or not.

```
SELECT f.id, f.fire_start, AVG(w.temp)
FROM Wildfires f, Parks p, Weather_History w
WHERE ST_Contains(p.boundary, ST_Make_Point(w.lat, w.lon))
    AND interval_overlapping(
        interval(f.fire_start, f.fire_end),
        w.reading_interval
    )
    AND ST_Distance(f.location, w.location) < 1
GROUP BY f.id, f.start;
```

Query 3.3: Interval and Spatial Join Query

Both Query 3.2 and Query 3.3 would likely end up processed by NLJ operators due to the limited availability of ready-to-use optimization tools for text-similarity and interval joins in most systems, even if we assume the data science team employed tools for spatial join queries for Query 3.1. In addition to that, note that Query 3.3 is a combination of both spatial and interval joins which makes it even harder to optimize. To the best of our knowledge, there is no DBMS today that would generate an optimized query plan for such queries.

3.1.2 A New Approach

We argue that if there were a straightforward way to implement and integrate optimized join algorithms into the query optimization engines for DBMSs, the availability of such optimizations would increase and the queries above could be efficiently processed. Consequently, data scientists would have more time and courage to extend their investigations.

In this work, we introduce the Flexible User-defined Distributed Joins (FUDJ) framework, which seeks to enhance the availability of optimized join algorithms within DBMSs. The FUDJ approach allows the implementation of partition-based distributed join algorithms without requiring in-depth knowledge of database internals or distributed programming while still achieving similar performance as if they were implemented as built-in operators inside a DBMS. Figure 3.1 shows where we are aiming to position FUDJ in comparison to the other implementation methods.

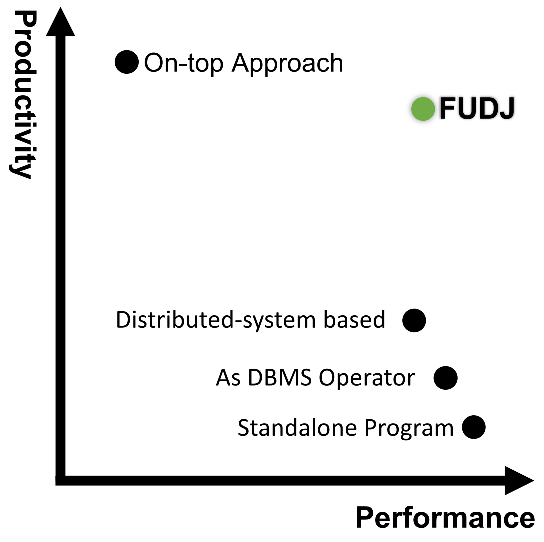


Figure 3.1: Productivity and Performance Evaluations of Existing Optimized Join Implementation Methods

To achieve these goals, we propose a novel extensibility architecture for implementing and incorporating join algorithms into a DBMS. Our approach involves identifying the fundamental principles shared among various distributed join techniques and integrating their touch points into the system's code base. The method is roughly similar to User-defined Aggregates (UDAs) [?], where users provide a function that accumulates the information into a state record and, in some cases, a combining step is utilized, especially in distributed or parallel processing scenarios, to efficiently combine partial results from different nodes or threads. Subsequently, a function is applied over the states to compute the actual aggregate value. We allow customization of the logic specific to each join operation through a series of specialized UDFs. In another sense, our approach is analogous to Generalized Search Trees (GiST) [40] to some degree. In GiST, the common logic, such as node merging and splits, is implemented in the code base of the database system itself. On the other hand, the index-specific logic, such as comparison operations within tree nodes, is defined by the developer creating a new index structure. In our approach, the developer defines the logic specific to each join operation. This specific logic is externalized through UDFs that encapsulate the join-specific logic, such as determining how the data will be partitioned and joined. This approach aims to strike a balance between efficiency and productivity, enabling the definition of new join operations with minimal lines of code (LOC) while maintaining high execution efficiency.

Our contributions can be summarized below.

- **FUDJ Programming Model:** We propose a new functional programming model that allows developers to implement existing and new partition-based distributed join algorithms without requiring database internal and distributed programming knowledge.
- **FUDJ Infrastructure:** Design of components to support FUDJ that could be applied to any DBMS with the following generic extensions:
  - Install join libraries with a "CREATE JOIN" statement,
  - Detect FUDJ queries and generate optimized query plans,
  - Offer a Serialization/Deserialization protocol that efficiently transfers tuples between the database engine and the FUDJ library.

- Realization of the concept on AsterixDB as proof of its feasibility, and providing FUDJ implementations for Spatial, Overlapping Interval and Text-Similarity Distributed Joins.
- Showing that the FUDJ implementations require roughly 20x less work while providing as much as 500x speed-up against on-top approaches, which is close to even advanced built-in implementations.

The rest of this chapter is organized as follows. Section 3.2 presents background on joins, query optimization, and database extensibility. Section 3.3 discusses related work. Section 3.4 addresses the commonalities and challenges of distributed optimized join algorithms. Section 3.5 presents our programming model while Section 3.7 provides details about our framework and its application to query optimizers. Section 3.6 shares the details of the realization of the architecture on AsterixDB and describes three example join algorithm implementations. Section 3.8 explores the performance of FUDJ, and Section 3.9 concludes our study and discusses future work.

3.2 Background

Join methods. Joining two dataset is an important and expensive task for any database system. Three commonly used join algorithms are hash joins, nested loop joins, and indexed nested loop joins. Nested Loop Join (NLJ) is the primitive approach for joining two dataset which is a basic two nested for loop where the two datasets scanned and the condition is applied to all tuple pairs. NLJ can be preferred especially if one of the dataset can fit into memory (Block Nested Loop Join). Another similar approach to NLJ is Indexed

Nested Loop Join (INLJ). INLJ can be used if the datasets are indexed already by pruning some of the records using the condition and the index resulting in using the condition less than the cartesian product. INLJ can be preferred if there is an index available for the key and the condition is applicable.

Nested loop joins operate by iterating through each record in one table and matching it with corresponding records in the other table based on the join condition. This algorithm is straightforward but can be resource-intensive for large datasets as it requires scanning the entire second table for every record in the first table.

Indexed nested loop joins leverage indexes to enhance the performance of nested loop joins. By utilizing indexes, the join process becomes more efficient, as it avoids scanning the entire second table and instead utilizes the index structure to locate matching records. This optimization significantly reduces the computational burden.

Hash joins involve building hash tables to efficiently match records from the joined tables. The join process entails hashing the join attribute values of one table and then probing the hash table with the join attribute values of the other table. This algorithm is particularly effective when dealing with large datasets and equality-based join conditions.

Query Optimization for Joins. Query optimization refers to the process of enhancing the performance of database queries by selecting the most efficient execution plans. When it comes to join queries, optimization becomes crucial due to the potential complexity and resource requirements of joining large tables. Optimizing join queries involves identifying the most suitable join algorithm, considering factors such as table sizes, available indexes, join conditions, and available system resources.

The optimization process aims to minimize the execution time and resource consumption of join queries. It involves analyzing query structures, estimating costs, and selecting the join algorithm that offers the best trade-off between resource usage and performance. Effective query optimization ensures that join operations are executed in an efficient and scalable manner, leading to faster query execution and improved overall system performance.

Database Extensibility. Database extensibility refers to the ability of a DBMS to be easily extended and customized to accommodate specific application requirements. This flexibility enables the integration of user-defined functionality, enhancing the capabilities of the database system. Three common aspects of database extensibility are UDFs, UDAs, and generalized index search trees.

UDFs allow users to define custom operations that can be executed within SQL queries. UDFs expand the range of available functions and provide users with the ability to tailor the database system to their specific needs. By incorporating UDFs into the query execution process, complex computations and data manipulations can be performed seamlessly within the database engine.

User-defined aggregates (UDAs) enable the creation of custom aggregate functions that can be applied to groups of data within a query. UDAs offer a powerful mechanism for performing specialized calculations and aggregations that may not be supported by standard SQL functions. These aggregates provide greater flexibility in summarizing and analyzing data based on specific user requirements.

Generalized index search trees extend the capabilities of traditional indexing structures by allowing the definition of custom search algorithms. These structures provide a way to optimize query execution for specialized data types and access patterns that are not adequately supported by standard indexing methods. By designing and implementing customized index search trees, database systems can achieve improved performance and efficiency in handling diverse data scenarios.

3.3 Related Work

Joins have been extensively researched both in academia and industry, spanning various domains. For instance, studies such as [38,46,72,107,115] propose methodologies for spatial joins, while survey papers such as [13,47,108,117] offer comprehensive evaluations of existing methods. Set-similarity joins have been optimized in studies such as [9,26,27,52,65, 73,85,93,95]. Trajectory joins are explored in papers such as [4,5,20,81,82,111], along with surveys such as [97]. JSON similarity studies are documented in [43,50]. Interval joins find attention in works like [12,17,28], while kNN joins are explored in [62,79]. It is important to note that each study introduces a method tailored for a specific join type. However, despite this rich literature, it remains challenging to assert that these methods are widely available within DBMSs. In other words, there is a scarcity of DBMSs that comprehensively support a diverse array of join types.

When it comes to **implementation methods for optimized joins**, these can be classified into three categories: distributed-system-based, as a DBMS operator, and standalone programs. Many of the aforementioned join studies opt for implementation through programming paradigms such as MapReduce [24], RDD [112], or PACT [8], or as standalone programs. However, these often lead to isolated solutions that cater to specific domains or systems, making their integration into a DBMS challenging. A select few approaches [17, 52, 72, 85] implement their methods within DBMSs. While these approaches advocate for the advantages of DBMS integration, their applicability to other optimized joins and DBMSs is limited, thereby necessitating a fresh implementation for each new join method.

In contrast, the concept of **database extensibility** has been a well-explored and well-established topic for decades [16]. Commonly adopted concepts include UDFs and UDAs. Moreover, the Generalized Search Trees (GiST) [40] introduces an extensibility framework that enables developers to implement and integrate custom indexing methods. Although GiSTs can enhance join performance in specific cases when paired with Indexed Nested Loop joins, they lack the capability to seamlessly integrate new join algorithms into a DBMS. Consequently, the concept of database extensibility has yet to encompass a method for accommodating *User-defined Joins*.

In summary, despite the rich existing literature for optimized joins, their availability in DBMSs and systems that can optimize a good variety of join types is limited. Also, the current preferred implementation methods for these optimized joins result in specialized programs which are far from being a universal model when it comes to the integration to DBMSs. Additionally, while the concept of database extensibility has seen advancements through mechanisms like UDFs, UDAs, and GiST, a comprehensive framework for accommodating User-defined Joins is missing.

3.4 Common Challenges in Distributed Join Processing

Efficient joining methods are essential for improving the data processing capabilities of distributed systems. The strategies employed in optimized joining methods are crucial for achieving this goal. In this context, three primary optimized join approach categories stand out: nested-loop joins, partition-based joins, and sort-merge-based joins. Nested-loop joins, though follow a straightforward method to distribute the data and are easy to implement, exhibit limited optimization potential due to their sequential nature, often resulting in suboptimal performance on large datasets. Sort-merge joins, while favorable if the data is already sorted and effective in parallelization for some cases, encounter challenges in shared-nothing environments due to the need for data shuffling across nodes, leading to increased network overhead.

On the other hand, partition-based joins exhibit promising potential by leveraging data partitioning and local processing, reducing data movement and network costs. These concepts lead to more parallelism and efficient utilization of resources, making partition-based methodology the most popular choice for optimizing joins in distributed systems resulting in numerous studies for various domains.

Consequently, as we aim to increase the availability of optimized joins in DBMSs, we introduce the FUDJ programming model that is designed to allow easy implementation of partition-based join algorithms on DBMSs. The key idea is identifying the common logic of partition-based distributed join techniques and injecting them into the code base of DBMSs while externalizing the logic related to specific join operations through user-defined join that are implemented using the FUDJ programming model.

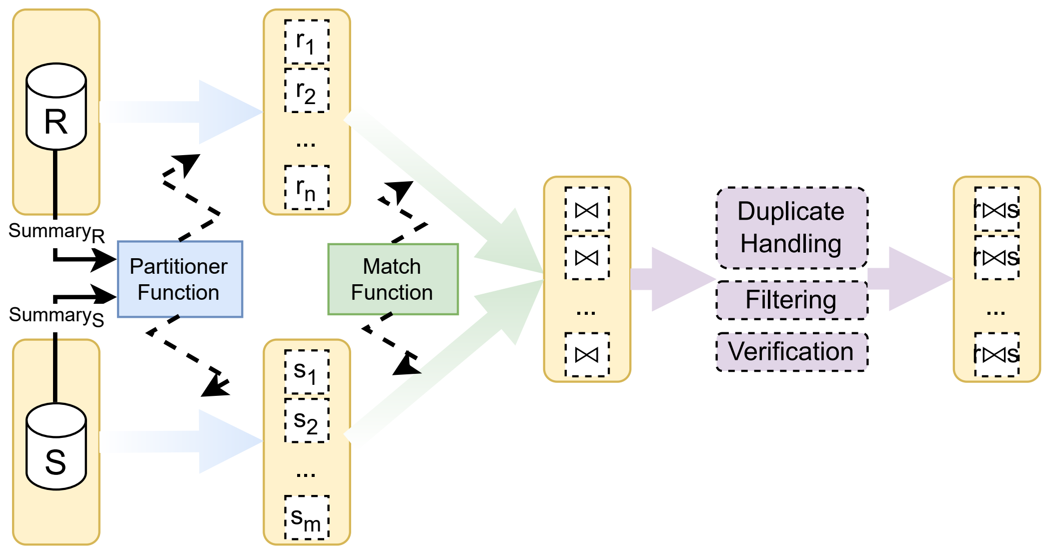
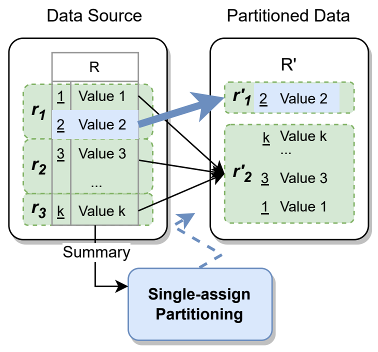


Figure 3.2: Summarize and Partition Phases

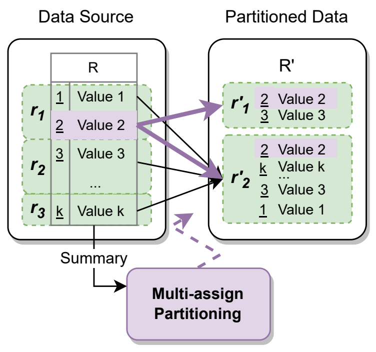
In the rest of this section, we identify the common challenges of partition-based joins. First, we divide the common challenges into two categories based on the natural phases of partition-based methods namely partition and join. As illustrated in Figure 3.2, in the partition phase, the data is partitioned in a way that the next phase, joining, requires the minimum resources and finishes the join process as efficiently as possible. Below we analyze each phase's challenges in its subsections.

3.4.1 Partitioning

The partitioning phase presents several challenges that require careful consideration [6]. One of the foremost challenges is achieving optimal data distribution across the nodes in the distributed system. Poor partitioning can result in data skew, where some nodes are overloaded due to unevenly distributed data. Moreover, identifying potentially matching keys is important to ensure that related data ends up on the same node, reducing the need for extensive inter-node communication during subsequent join operations. Balancing partition granularity and size is yet another challenge. Overly fine-grained partitions might lead to excessive overhead, while coarse-grained partitions could affect parallel processing efficiency. Addressing these challenges in the partitioning phase is paramount for achieving a well-balanced, efficient, and scalable partition-based approach within distributed systems. To ensure optimal performance and overcome these challenges, it is crucial to have a thorough comprehension of data characteristics. As shown in Figure3.2, an initial scan of the input dataset to collect such information (Summary) to have a better partitioning is one the most common ways. For instance, OIPJoin algorithm [28], requires minimum interval start, and maximum interval end times to divide the space into equal sized granules. PBSM [72] computes the Maximum Bounding Rectangles (MBR) of the input and divides it into tiles. Finally, Text-simililartiy join approach described in [53] counts the words from input datasets and sort them by their ranks to find out the least common words in each records. In all these scenarios, the input space is divided into **buckets** at the logical level, and each records is assigned to a physical partition accordingly by relying on buckets.



(a) Single-assign



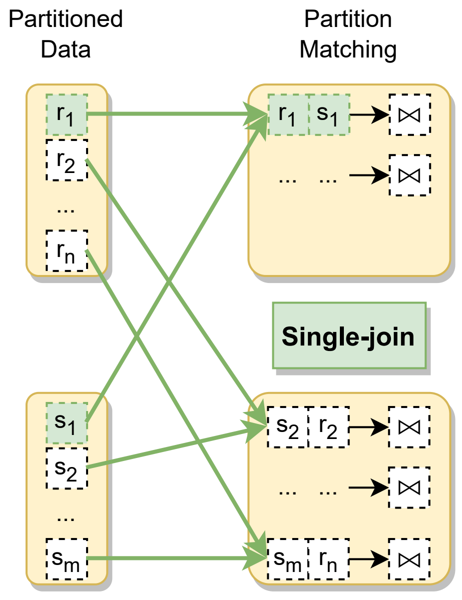
(b) Multi-assign

Figure 3.3: Partitioning Categories

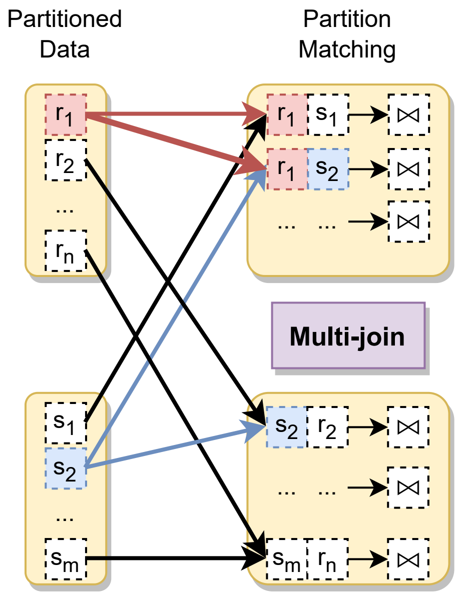
Furthermore, it is important to note that some partitioning approaches result in data replication (**multi-assign**) across partitions while others do not. Replication can help mitigate data skew and reduce inter-node communication during joins, but it comes at the cost of increased storage overhead. For instance, PBSM [72] assigns each geometry to all of the tiles that they are overlapping with. Text-similarity join [53] assigns the text to a certain amount of token based on the ranking of the words and the similarity threshold. Non-replicative strategies (**single-assign**), on the other hand, focus on maintaining unique sets of data on each node, reducing storage overhead but requiring more careful load balancing and efficient data movement during joins. For instance, OIPJoin [28] assings intervals to the smallest interval bucket that it can fit. In the rest of this chapter, we will term the stage where the algorithms collect information about the data in the partitioning phase as **SUMMARIZATION**, and the **SUMMARY** will be used to refer to the data where the information is stored. Since the SUMMARY from both sides of the joins is collected to conclude a partitioning strategy, the final information should be stored and available for the joining phase. We will refer to this state as **CONFIG** and the method that divides the input space into granules for the partitioning as **divide**. Finally, the methods that assign records to partitions by using CONFIG will be termed as **assign**, and the identifiers of the records to determine their partitions will be termed as **bucket**. Please note that a partition can contain one or more buckets.

3.4.2 Joining

One of the primary challenges in the joining phase is the task of matching the partitions. Eliminating irrelevant partitions from consideration or moving the relevant partitions to the same nodes before the join operation can reduce unnecessary data movement and processing. The matching method for the partitions plays a crucial role to have an efficient strategy to have an efficient partition matching. We can categorize such methods into two: single-join (See Figure 3.4a) and multi-join (See Figure 3.4b). **Single-join** here means each partition can only match with a single partition thus it becomes an equijoin operation where the partitions can be hash partitioned as we see in hash-based joins. For instance, PBSM [72] only joins the records that overlaps with the same tiles, and in Text-similarity join [53], the records shares the same tokens are matched only. In a **multi-join** strategy, on the other hand, a partition can match with one or more buckets which makes it a theta-join operation. As a result, partitions from one side are mostly broadcasted. OIPJoin [28] is an example of that since one interval partition can match with multiple partitions.



(a) Single-join



(b) Multi-join

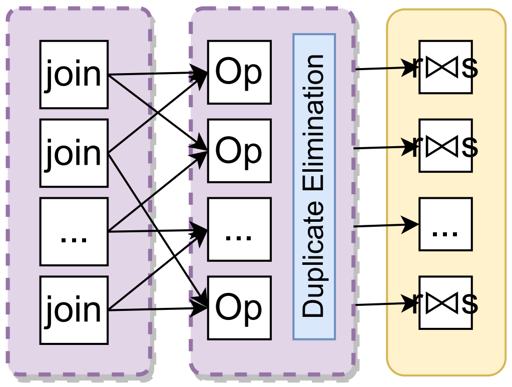
Figure 3.4: Partition Matching Strategies

Local optimization strategies are also applied during the joining of partitions in each node. Implementing a customized efficient join algorithm within individual partitions or nodes to minimize the computational and memory overhead or utilizing parallel processing techniques within partitions to take full advantage of available processing power and speed up the join operation is common to see in partition-based join algorithms. In cases of unbalanced partitions, memory utilization can become problematic too. Some partitions might not fit entirely in memory, requiring the utilization of memory budget-aware operators that can spill to the disk. Another optimization can be sorting. By sorting tuples within partitions based on join attributes, merge join algorithms can be utilized, requiring a smaller memory footprint.

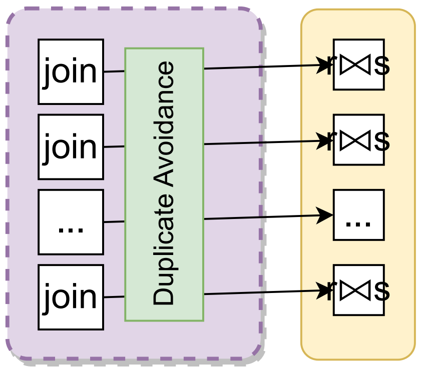
In addition, partitioning strategies that involve duplicating tuples across multiple partitions can introduce duplicate handling challenges. Duplicate elimination becomes essential in the subsequent stages. Duplicate elimination involves identifying and eliminating duplicate tuples from the joined output. Avoidance techniques, on the other hand, aim to prevent duplicates during the join process itself by cleverly designing matching and partitioning strategies.

At the conclusion of the join phase, the filtering and verification stages come into play. Filtering involves eliminating tuples that do not satisfy the join condition. Verification, on the other hand, ensures that all tuples that should be in the join result are indeed present.

In the FUDJ programming model, we will refer to the methods that are used to match the buckets as **match**. The logic of the match function actually defines whether the join is a single-join or multi-join. For instance, if the match is a simple equality, then



(a) **Duplicate Elimination**



(b) **Duplicate Avoidance**

Figure 3.5: Partitioning Categories

the join becomes a single-join and the system can utilize the optimized hash join operator. Lastly, the function that verifies the tuple pairs to finalize the join operation will be called **verify**. The verify function usually uses the **CONFIG** from the partitioning phase to determine whether the tuple pair belongs to the output or not.

3.5 Programming Model

Based on the common challenges and solutions in partition-based distributed optimized join algorithms that we describe in Section 3.4, we propose the FUDJ functional programming model that consists of three phases namely, **SUMMARIZE**, **PARTITION**, and **COMBINE**.

Figure 3.6 shows all the functions within its phase. We provide more details about the phases and the functions in the following subsections.

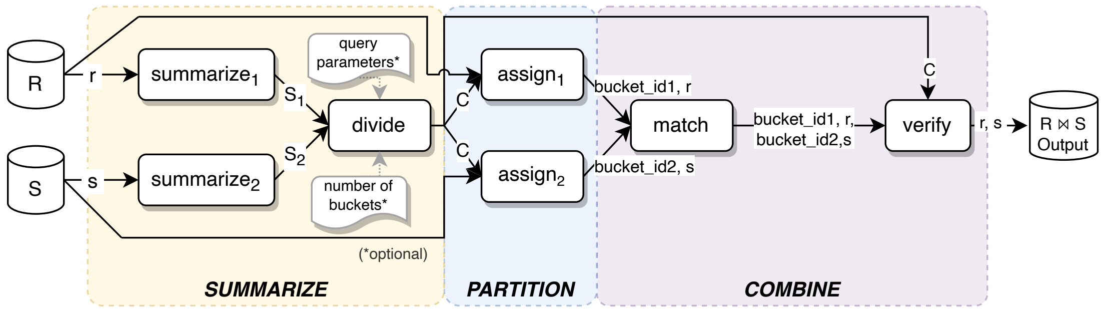


Figure 3.6: Flexible User-defined Distributed Join Data Flow Diagram

3.5.1 SUMMARIZE

To successfully decide how to partition the datasets, the join algorithms usually apply an initial step where the datasets are first analyzed, and summarized to have a better partitioning in the later phases. The summary can be the minimum bounding rectangle for a spatial dataset [72], minimum starting and maximum ending time for an interval dataset [28], or word frequencies for text-similarity joins [93]. Since we are considering distributed systems, independent from the strategy, first, all the records should be visited locally, then the local information should be merged and concluded into one global summary. This is a typical aggregation operation that we can achieve by first running a local aggregation and then merging it with a global aggregation.

We first provide two aggregate function interfaces as below.

$$local\_aggregate(T\,key, SUMMARY\langle T\rangle S) : SUMMARY\langle T\rangle$$

$$global\_aggregate(SUMMARY\langle T\rangle\, S) : SUMMARY\langle T\rangle$$

The $local\_aggregate$ function is used to read keys from input dataset and create $SUMMARY$ objects locally. Then all $SUMMARY$ objects merged into global $SUMMARY$ objects by *global_aggregate* function. Please note that, the framework allows to have two versions of local and global summary functions since key types can be different. On the other hand, if the keys have the same type and only one version is implemented, the optimizer will be aware of that and apply relevant optimizations if the query is for a self join.

Lastly, to divide the input domain space into meaningful partitions, we provide $DIVIDE$ function that takes two global $SUMMARY$ from both sides of the join and return a $CONFIG$ object.

$$divide(SUMMARY\langle T\rangle\ S_1, SUMMARY\langle T\rangle\ S_2) : CONFIG$$

For instance, *divide* combines two MBRs from both sides and returns the final MBR and carries the number of tiles for spatial join, and minimum start time, maximum end times for both datasets, and number of interval partitions for interval join, and ordered token rank list which is computed by using word counts from both sides for text-similarity join.

3.5.2 PARTITION

The goal of this phase is to assign the input datasets into subgroups which we will call as *bucket* and represent with a unique integer *bucket_id*. The framework, will than use the list of $bucket_id$ and the logic of the join algorithm to decide how to partition the input datasets. For spatial join, a *bucket* is a tile and $bucket_id$ is the tile_id, and for text-similarity join, a *bucket* is a word from word count list and $bucket_id$ is the rank of that word.

$$assign(T\,key, CONFIG) : int[]$$

After the SUMMARIZE phase, the input datasets will be scanned and for each *key*, *assign* function will return a list of *bucket_id* which is computed based on $CONFIG$. A *key* can be assigned to only one *bucket* (Single-Assign) or multiple buckets (Multi-Assign). In addition to that, for the cases where different *key* types from different side of the join, there can be a second implementation of *assign* function.

3.5.3 COMBINE

In this phase, first, we need to determine which *bucket* matches with which *bucket* by using the *match* function. As we mentioned previously we have two cases for this stage: single-join or multi-join. For single-join algorithms, we provide a default *match* function which basically checks whether the both $bucket_id$ is the same or not. For this type of algorithms, the developer should just use the default implementation since further optimizations can be applied.

$$match(int\, bucket\_id1,\, int\, bucket\_id2) : boolean$$

After we match the buckets, the next step is verifying the record pairs by using the verify function.

$$\begin{aligned} verify(&int\, bucket\_id1, T\, key1\\ &int\, bucket\_id2,\, Tkey2,\\ &CONFIG\, C) : boolean \end{aligned}$$

FUDJ's default duplicate avoidance method relies on utilization of the *assign* functions with $CONFIG$, and producing the list of $bucket_id$ for each record pair to find out if the matching buckets are the first matching pair or not. As we discussed in Section 3.4, some algorithms yield duplication due to the assignment of records to multiple buckets. FUDJ framework takes care of the duplicates in the verify phase. For the algorithms that do require another method for deduplication they can easily override the verify function provided by the framework or if there is no need for the deduplication developers just disable that to have a more efficient query processing.

3.6 Example Implementations

In this section we provide three example FUDJ implementations for spatial, text-similarity and overlapping interval joins. These examples are representing the FUDJ versions of the algoritmhs that we studied in Section 3.4.

3.6.1 Spatial FUDJ

Our Spatial FUDJ implementation is based on PBSM algorithm described in [72]. We start with calculating the MBRs of each dataset with the *summarize* function. Here, $MBR()$ function returns the MBR of the given *geometry* and $\cup$ merges two MBRs and returns an MBR that covers both MBRs.

```
1: function SUMMARIZE($geometry$, $SUMMARY$)
2:     $SUMMARY \leftarrow MBR(geometry) \cup SUMMARY$
3:     return $SUMMARY$
```

After we have MBRs from both side of the join, we then use *divide* function to compute the final MBR and create the grid that divides the space into $n \times n$ bucket. Next, we store the MBR and n into $CONFIG$.

```
1: function DIVIDE($SUMMARY_1$, $SUMMARY_2$, $n$)
2:     $MBR \leftarrow (SUMMARY_1 \cup SUMMARY_2)$
3:     $CONFIG \leftarrow (MBR, n)$
4:     return $CONFIG$
```

Now, our spatial join algorithm can assign geometries to relevant buckets. To simplify the algorithm here the function *getOverlappingTileIds*() represents a function that divides the space into $n \times n$ equal-sized tiles and returns the ids of the tiles that overlap with the given geometry's MBR.

```
1: function ASSIGN($geometry$, $CONFIG$)
2:     $MBR \leftarrow MBR(geometry)$
3:     $tileIDs \leftarrow getOverlappingTileIDs(MBR, CONFIG)$
4:     return $tileIDs$
```

When it comes to matching buckets, since our algorithm follows single-joining strategy, we utilize the default *match* function. Finally, we provide the simple $verify$ function that only checks if the actual geometries are intersecting or not below.

```
1: function VERIFY($geometry_1$, $geometry_2$)
2:     return $intersects(geometry_1, geometry_2)$
```

3.6.2 Text Similarity FUDJ

Similar to [52], we first count the words of all the records in the summary step by using a hash map. Here the *tokenize*(*text*) function is used to get the list of the words of each text.

```
1: function SUMMARIZE($text$, $SUMMARY$)
2:     $tokens \leftarrow tokenize(text)$
3:     for each $token \in tokens$ do
4:         $SUMMARY.merge(token, 1)$
5:     return $SUMMARY$
```

In divide, we first combine the two hash maps that consists the number of occurrence of each word from both sides. Next, *sortByCount*() function sorts the words by their counts in descending order and returns a new hash map that has the rank of each word as value. Finally, the word rank map is put into the configuration along with the similarity threshold.

```
1: function DIVIDE($SUMMARY_1$, $SUMMARY_2$, $SimilarityThreshold$)
2:     for each $token \in SUMMARY_2$ do
3:         $SUMMARY_1.merge(token, SUMMARY_2.get(token))$
4:     $TokenRanks \leftarrow sortByCount(SUMMARY_1)$
5:     $CONFIG \leftarrow (TokenRanks, SimilarityThreshold)$
6:     return $CONFIG$
```

In assign, we first create the sorted rank list of words for each text. Then, we calculate the prefix length p for each text using the similarity threshold. Finally we assign the text to the buckets that are defined by the first p ranks of each text.

The purpose of this method is assigning each text to the smallest possible buckets and choosing the rarest words of each text to increase the pruning.

```
1: function ASSIGN($text$, $CONFIG$)
2:     $tokens \leftarrow tokenize(text)$
3:     $tokenRanks \leftarrow \emptyset$
4:     for each $token \in tokens$ do
5:         $tokenRanks.add(CONFIG.TokenRanks.get(token))$
6:     $l \leftarrow len(tokens)$
7:     $prefixLength \leftarrow (l - ceil(C.SimilarityThreshold * l)) + 1$
8:     $bucketIds \leftarrow copyRange(sort(tokenRanks), prefixLength)$
9:     return $bucketIds$
```

Finally, in verify we calculate the jaccard similarity of the two sides as below and return true if they are above the threshold.

```
1: function VERIFY($text_1$, $text_2, CONFIG$)
2:     $threshold \leftarrow CONFIG.SimilarityThreshold$
3:     $tokens_1 \leftarrow tokenize(text_1)$
4:     $tokens_2 \leftarrow tokenize(text_2)$
5:     $similarity \leftarrow (|tokens_1 \cap tokens_2| \div |tokens_1 \cup tokens_2|)$
6:     return $similarity > threshold$
```

3.6.3 Overlapping Intervals FUDJ

To partition the date first we need to divide the space into granules. For that purpose, we start with finding the minimum start and maximum end times of each side with summarize function.

```
1: function SUMMARIZE($interval$, $SUMMARY$)
2:     if $SUMMARY = NULL$ then
3:         $SUMMARY \leftarrow (+\infty, -\infty)$        ▷ Init. Min Start and Max End Time
4:     if $interval.start < SUMMARY.minStart$ then
5:         $SUMMARY.minStart \leftarrow interval.start$
6:     if $interval.end > SUMMARY.maxEnd$ then
7:         $SUMMARY.maxEnd \leftarrow interval.end$
8:     return $SUMMARY$
```

In divide function, we first combine two summaries and unify both time spaces. Next, we divide the space into $NumberOfBuckets$ bucket and calcualte the length of each bucket. Finally, we put all the information required to assign records to the partitions together into $CONFIG$.

```
1: function DIVIDE($SUMMARY_1$, $SUMMARY_2, NumberOfBuckets$)
2:     if $SUMMARY_2.minStart < SUMMARY_1.minStart$ then
3:         $SUMMARY_1.minStart \leftarrow SUMMARY_2.minStart$
4:     if $SUMMARY_2.maxEnd > SUMMARY_1.maxEnd$ then
5:         $SUMMARY_1.maxEnd \leftarrow SUMMARY_2.end$
6:     $Range \leftarrow SUMMARY_1$
7:     $length \leftarrow (Range.maxEnd - Range.minStart)$
8:     $d \leftarrow length/NumberOfBuckets$
9:     $CONFIG \leftarrow (Range, d, NumberOfBuckets)$
10:    return $CONFIG$
```

Each interval needs to be assigned to the smallest bucket that it can fit. By using the length of each granule and minimum start time of the space, we find the starting and ending granule ids for each interval. Then we combine these two ids into one integer as bits.

```
1: function ASSIGN($interval, CONFIG$)
2:     $R \leftarrow CONFIG.Range$
3:     $d \leftarrow CONFIG.d$
4:     $front \leftarrow (interval.start - R.minStart)/d$
5:     $end \leftarrow (ceil(interval.end - R.minStart)/d) - 1$
6:     $bucketId \leftarrow (front << 16)|(end\&0xFFFF)$
7:     return $bucketId$
```

Bucket matching is not equality. So, we need to implement a match function that first extract the starting and ending granule ids of each bucket, and return true if buckets are overlapping.

```
1: function MATCH($bucketId_1, bucketId_2$)
2:     $b1Start = bucketId_1 >> 16$
3:     $b1End = (short)bucketId_1;$
4:     $b2Start = bucketId_2 >> 16$
5:     $b2End = (short)bucketId_2$
6:     return $(b1Start \leq b2End)$ and $(b1End \geq b2Start)$
```

Finally, we test i_1 and i_2 to see if they are overlapping or not in the verification phase.

```
1: function VERIFY($i_1, bucketId_1, i_2, bucketId_2, CONFIG$)
2:     return $(i_1.start < i_2.end)$ and $(i_1.end > i_2.start)$
```

3.7 FUDJ Infrastructure

In this section, we present the components of the FUDJ Framework. In general, the framework can be implemented in any DBMS since it relies on the common concepts of built-in functions, UDFs, and rule-based query optimization. We divide this section into three subsections. First, we talk about how the logic from external join libraries will be linked into the system through proxy built-in functions. Second, we will explain how new join algorithms can be registered through a novel statement "CREATE JOIN". Finally, we describe how can the DBMSs utilize FUDJs and generate optimized query plans using rewrite rules.

3.7.1 Internal and External Actors

UDFs are well-known components of modern DBMSs, allowing users to implement custom functions and integrate them into their system to process their data. With UDFs, complex join predicates can be implemented, and various join operations can be performed. However, it is important to note that UDFs are primarily supported as scalar functions, and queries using UDFs may not achieve the same level of performance as those employing optimized join algorithms due to be processed by NLJ operators.

Similarly, UDA functions assist developers in creating their custom aggregate functions. UDAs distinguish themselves from UDFs by having a state that is used for aggregation and by involving stages such as local and global aggregation. In both cases, database management systems typically define a distinct type of function. From the outset of query optimization, DBMSs associate external library information with the function signature,

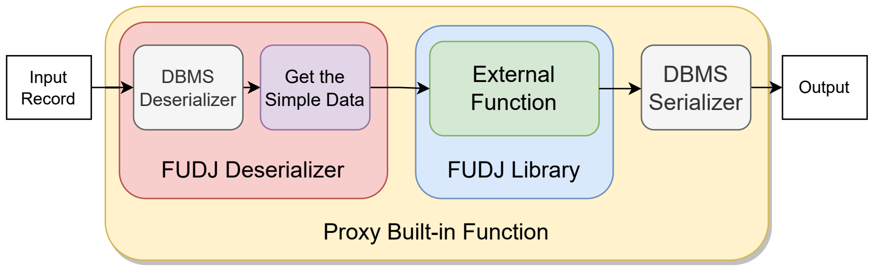


Figure 3.7: A Proxy Built-in Function in FUDJ Framework

utilizing these libraries at runtime.

In the FUDJ Framework, we follow a similar principle. For each function interface within our programming model, we provide a corresponding built-in function implemented internally as internal actors. We also introduce a new external function signature type associated with the FUDJ framework. When a new join algorithm is created, the FUDJ framework generates FUDJ-specific UDF signatures, which include the join library information for all functions in the programming model. These signatures are then registered with the system as external actors. During runtime, whenever the DBMS encounters an external actor call with the FUDJ's external function signature, it must modify the evaluator using the information embedded in the signature. Subsequently, it creates the internal actor evaluator by passing the external FUDJ library information. Then in each internal actor, FUDJ library should be initiated as an object only once.

In each built-in function, DBMSs deserialize records before processing. Most DBMSs internally implement data types for various data types with specific serialization and deserialization methods. For example, Apache AsterixDB has specific type handling internally for data types like "AInt" for integers. However, in FUDJ, as the programming model is designed to work with simple data types, an additional step is required to convert DBMS-specific data into simple data types. Figure 3.7 shows how the data transfer works internally in a proxy built-in function of FUDJ. It is worth noting that some types require specific handling; for instance, intervals can be converted into long arrays, where the first element represents the start time and the second the end time. This aspect of the framework is critical and requires careful implementation to avoid excessive overhead during runtime. However, it is not a very expensive step as the only requirement is retrieving the data from the object that is already deserialized as we show with evaluations in Section 3.8.

As discussed in Section 3.5, we have two states to consider: $SUMMARY$ and $CONFIG$. Since DBMSs already have solutions for built-in aggregate functions, we only need to adhere to existing design principles and handle $SUMMARY$ as a regular state within a typical aggregate function. The same principle applies to $CONFIG$, which can be treated as a single record with its type set as "Object." This approach also simplifies state transfer, as both states appear as regular records from the database perspective.

3.7.2 Query Optimizer Integration

The first task of the query optimizer is to determine whether the join query includes a FUDJ predicate. This detection is accomplished by examining the predicate function signature. When a FUDJ predicate is detected, the query optimizer retrieves the external library information from the metadata and commences the generation of the join query plan, as depicted in Figure 3.8. For each stage in the FUDJ query plan, the optimizer creates corresponding FUDJ external function calls.

During runtime, as mentioned in the preceding section, each external FUDJ function undergoes modification to incorporate the related proxy built-in function, and external library information is associated with it. The query optimizer must also apply physical optimizations when applicable. In this initial design, we introduce two further optimizations. The first one pertains to self-joins. Typically, DBMSs optimize self-joins by replicating intermediate results that are used multiple times during query processing. For instance, in a Spatial self-join, the resulting MBR (Minimum Bounding Rectangle) of one side after the summarization stage can be replicated and fed into the $DIVIDE$ function since the MBR computation is the same for both sides. Query optimizers are usually designed to handle self-joins in this manner. Consequently, the only requirement for the FUDJ framework is to detect whether FUDJ implements separate $SUMMARY$ and $PARTITION$ stages or not. This can be achieved by checking if the FUDJ is overriding the default *summary* and *assign* functions. If the FUDJ uses the default functions, the same function signature is used, enabling the query optimizer to apply further optimizations.

The second optimization concerns selecting the appropriate join operator for bucket matching. For single-join FUDJs with a bucket matching condition as equality, the optimizer can employ the Hash Join operator. This is advantageous, as Hash Partitioning can also be applied. Similar to the previous approach, the optimizer must check if the 'match' function is overridden or if it is using the default implementation to apply further optimization by compelling the DBMS to utilize the Hash Join operator and partitioning, as illustrated in Figure 3.9.

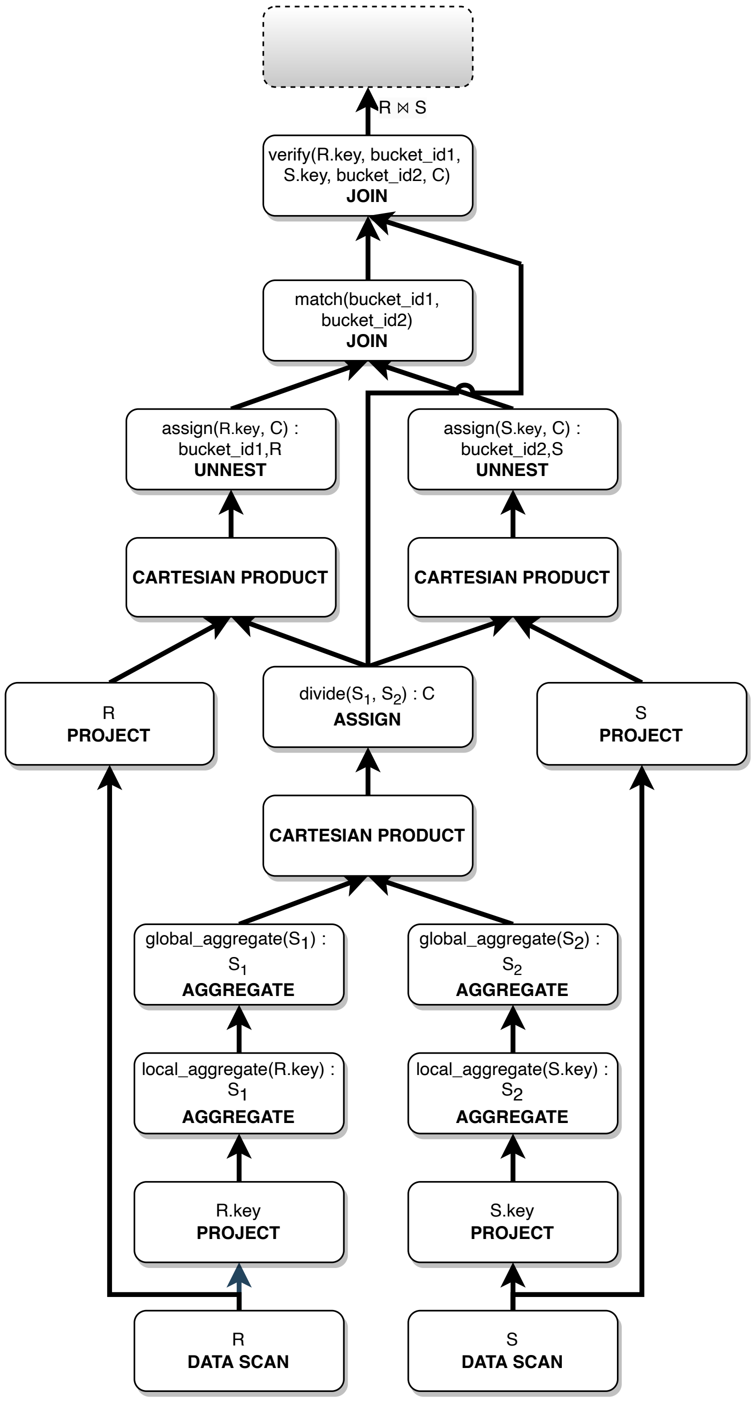

R ⋈ S
verify(R.key, bucket_id1, S.key, bucket_id2, C)
JOIN
match(bucket_id1, bucket_id2)
JOIN
assign(R.key, C) : bucket_id1,R
UNNEST
assign(S.key, C) : bucket_id2,S
UNNEST
CARTESIAN PRODUCT
CARTESIAN PRODUCT
R
PROJECT
divide(S1, S2) : C
ASSIGN
S
PROJECT
CARTESIAN PRODUCT
global_aggregate(S1) : S1
AGGREGATE
global_aggregate(S2) : S2
AGGREGATE
local_aggregate(R.key) : S1
AGGREGATE
local_aggregate(S.key) : S2
AGGREGATE
R.key
PROJECT
S.key
PROJECT
R
DATA SCAN
S
DATA SCAN

Figure 3.8: Flexible User-defined Distributed Join Logical Plan

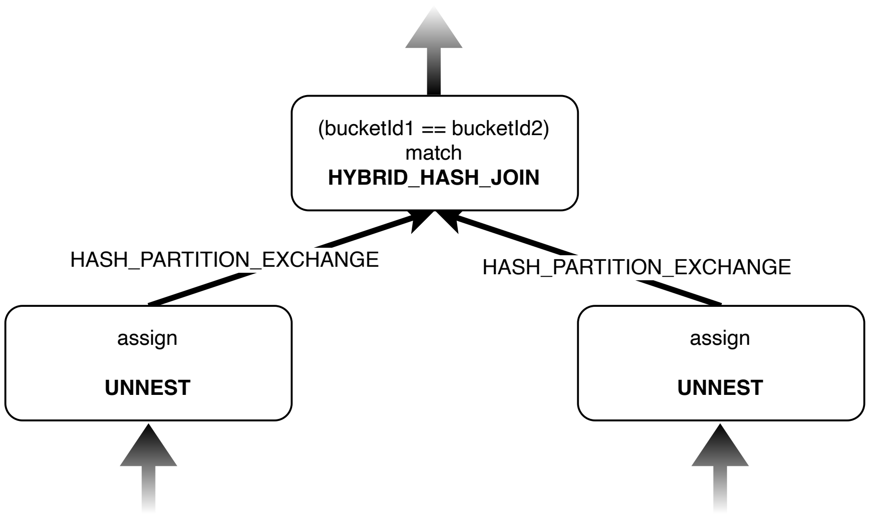


Figure 3.9: Single-Join Algorithms - Bucket Matching Phase

It is important to note that since the query optimizer generates query plans for FUDJ join queries as part of its overall optimization process, FUDJ query processing can take advantage of all the optimizations applied by the optimization engine. For example, if the join query involves filtering operations, the optimizer will prioritize executing them before the join query plan. Similarly, if there is a group by operator in the query, the optimizer can generate efficient query plans to handle that part of the operation

3.7.3 Creating Joins

As outlined in our goals, our aim is to facilitate easy and convenient installation of joins. To achieve this, we introduce a new SQL statement called "CREATE JOIN." In Query 3.4, we provide an example of creating a join named "text_similarity_join" with two "String" keys. The external logic for this join is sourced from the "flexiblejoins" library, with the package and class name set as "setsimilarity.SetSimilarityJoin." Notably, this join has three parameters. In this specific example, the join is based on a similarity metric, and a predicate is considered satisfied if the metric surpasses a specified threshold. Given that the algorithm necessitates the threshold in all stages (including prefix filtering), this information is embedded into the caller function's signature.

```
/*Creating a FUDJ*/

CREATE JOIN text_similarity_join(a:string, b:string, t:double) RETURNS boolean

AS "setsimilarity.SetSimilarityJoin"

AT flexiblejoins;

/*Dropping a FUDJ*/

DROP JOIN text\_sim\_join(a:string, b:string, t:double);
```

Query 3.4: Create Text-similarity Join

After executing Query 3.4, the DBMS creates all the corresponding UDFs and registers the library information for them. When it comes to removing a join, similar to other operations, we only need to run 'DROP JOIN text_similarity_join(a: string, b: string, t: double),' and all UDFs will be removed.

3.7.4 Realization of the Infrastructure

We implement the FUDJ framework on Apache AsterixDB [?, 2], and provide a simple standalone version for testing purposes. Also, the first FUDJ interface based on the FUDJ programming model is implemented in Java language to make it compatible with Apache AsterixDB.

AsterixDB Implementation

Apache AsterixDB [2] is an open-source, scalable Big Data Management System (BDMS) that offers a flexible data model, distributed storage and transactions, rapid data ingestion, and data-parallel query execution runtime. In this section, we briefly describe how we implemented the FUDJ Framework on Apache AsterixDB by adhering to the implementation guidelines.

Apache AsterixDB offers a wide range of built-in functions. It also support UDFs and allows developers to implement their custom functions. However, it is required to use Apache AsterixDB specific data types. We further improved this feature for FUDJ and let the developer use Java primitive types. In addition to that, while there are built-in aggregate functions available in the codebase, currently, there is no support for external aggregate functions. Therefore, we had to modify its runtime mechanism to accommodate external aggregate functions. External functions are connected to Apache AsterixDB through the concept of libraries, and we utilized the same mechanism for FUDJs as well. In contrast to the default one-to-one mapping between external functions and libraries, in our case, one library comprises all the functions belonging to the FUDJ programming model, and all UDFs created for FUDJ are assigned to that library.

Query optimization is done in Apache AsterixDB by incorparating a set of predefined rules that dictate how queries should be transformed and optimized. We implemented a rewrite rule that checks the condition of the join query and intervenes if the join condition involves a FUDJ function. Then, the rule is building the query plan by following the steps described previously.

Standalone (Single-Machine) Version

One of the biggest challenges for joining algorithm integration into DBMSs is debugging and testing due to the complex notion of DBMSs. Having strict mechanisms for query processing and data reading makes it hard to handle bugs or test new ideas easily without rebuilding or redeploying the system. Motivated by these challenges, we also provide a Single-machine Standalone version of the FUDJ Framework. The standalone version can run any FUDJ algorithm for testing and debugging purposes. Since it simply reads the data and feeds it to the FUDJ, finding the logical bugs or trying new ideas is straightforward. We share JAVA implementation with this study, but it can also be transformed into another programming language easily.

3.8 Experiments

In this section, we evaluate FUDJ framework and Spatial, Interval and Text-similarity FUDJ implementations applied to Apache AsterixDB. Our evaluation begins with a productivity assessment of the implementation methods (FUDJ and built-in). Next, we demonstrate that the utilization of the FUDJ extensibility framework introduces negligible to zero query processing overhead when compared to the built-in approach. We then proceed to evaluate the performance and scalability of the three example join implementations in comparison to on-top solution (NLJ operator with a UDF). Finally, we study alternative duplicate handling strategies and outline future directions for the FUDJ framework and programming model by comparing it against advanced optimized join implementations.

Hardware setup: The experiments run on a cluster with one head node and 12 worker nodes. The head node has Intel(R) Xeon(R) CPU $E5-2609$ v4 @ 1.70GHz processor, 128 of GB RAM, 2 TB of HDD, and 2×8-core processors running CentOS and Java 17.0.1. The worker nodes have Intel(R) Xeon(R) CPU E5-2603 v4 @1.70GHz processor, 64 GB of RAM, 10 TB of HDD, and 2×6-core processors running CentOS and Java 17.0.1.

Datasets: We use four real-world datasets. For spatial join queries Parks [31] and Wildfires [86] datasets are used , NYCTaxi [101] is used for interval join queries, and AmazonReview [39] is used for text-similarity queries.

Table 3.1: Datasets for FUDJ Experiments

Name	Size	#Records	Key Type
Wildfires [86]	22.1 GB	18M	Point
Parks [31]	7.7 GB	10M	Polygon
NYCTaxi [101]	38.8 GB	173M	Interval
AmazonReview [39]	58.3 GB	83M	Text

Implementations: We implemented **FUDJ framework** on Apache AsterixDB version 0.9.8. The three example join algorithms Spatial, Interval, and Text-similarity that are based on studies [28, 53, 72] are implemented on Apache AsterixDB from scratch, and we will refer to them as **built-in** implementations. Finally, we implemented the **FUDJ versions** of the three example join algorithms and installed them on Apache AsterixDB. We will use the term **on-top** to refer to join query processing using the BNLJ operator in Apache AsterixDB.

Workload(Queries): We evaluate join implementations by using the queries from Query 3.5. Spatial join query counts the number of wildfires that are occurred in each park. Text-similarity join query computes the jaccard similarity of each review pair that have overall ratings 4 and 5 and counts the ones that are similar. Overlapping interval join query finds overlapping taxi rides belongs to different vendors. For each experiment, we stop query processing after 4000 seconds and assume the setup is not scalable for processing the query.

```
/*Spatial Join*/
SELECT p.id, count(1) c FROM Parks p, Wildfires w
WHERE ST_CONTAINS(p.boundary, w.location)
GROUP BY p.id
/*Text-similarity Join*/
SELECT COUNT(1) FROM AmazonReview r1, AmazonReview r2
WHERE r1.overall = 5 AND r2.overall = 4 AND
    similarity_jaccard(word_tokens(r1.review), word_tokens(r2.review)) >= 0.9;
/*Interval Join*/
SELECT COUNT(1) FROM NYCTaxi n1, NYCTaxi n2
WHERE n1.Vendor = 1 AND n2.Vendor = 2 AND overlapping_interval(n1.ride_interval,
    n2.ride_interval);
```

Query 3.5: Queries for the experiments

3.8.1 Productivity

Since both FUDJ and Built-in versions implement the same algorithms, we use Lines of Code (LOC) as a metric for productivity evaluations. For built-in implementations, we implement all required built-in functions and a rewrite rule for query optimization. On the other hand, as we explained in Section 3.5, FUDJ framework empowers the developer to define the logic for each function, allowing for flexibility and customization while significantly reducing the LOC required. Figure 3.10 clearly illustrates that FUDJ versions of the Spatial, Interval, and Text-similarity joins demand significantly fewer LOC, highlighting the efficiency and developer-centric design of the framework and the programming model. Note that here we are not comparing FUDJ against the distributed-system-based approaches since the yielding applications cannot be integrated directly into DBMSs hence they require the same lengthy process as in built-in approaches.

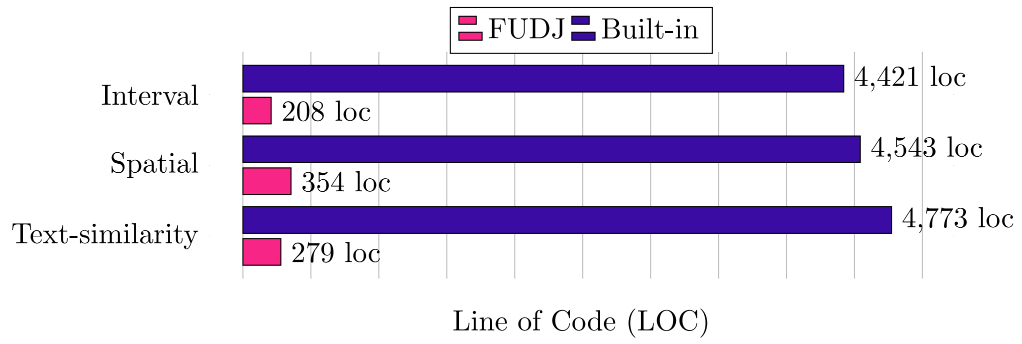


Figure 3.10: Lines of Code (LOC) Comparison of Join Implementations Using FUDJ and Built-in Approaches

Moreover, the reduced LOC in FUDJ versions not only improves productivity but also simplifies the debugging testing and code reviewing processes. With fewer LOC to manage, developers can pinpoint issues more easily and expedite the debugging phase. Additionally, since users have control over the logic of each function within FUDJ, they can fine-tune and adapt their code for specific testing scenarios, enhancing the robustness and reliability of their applications. This combination of reduced LOC and enhanced debugging and testing capabilities underscores the advantages of the FUDJ framework in distributed programming and database internals.

The integration of new join algorithms into traditional DBMSs often incurs significant deployment costs. Following the finalization of the implementation, it is common practice to rebuild the DBMS software. In our experimental environment, rebuilding a DBMS like AsterixDB typically takes around 5 minutes, which, although a necessary step, can introduce delays. However, in distributed systems, the process becomes more intricate, as the rebuilt package must be deployed to each node, consuming additional minutes. Furthermore, the DBMS often necessitates stopping and rerunning, causing further disruptions. In contrast, FUDJ offers a distinct advantage in this regard. It eliminates the need for such extensive deployment procedures, as new FUDJ packages can be swiftly and seamlessly deployed within seconds without any disruption to the system, making it a more efficient choice for introducing new join algorithms.

3.8.2 Performance

Figure 3.11 shows the evaluation of the three implementation methods run on 12-core for a variety of the data sizes. Here we run queries using subset of the datasets to control the workload. For Spatial FUDJ, the number of buckets, which is equivalent to the grid size that divides the space into tiles is set to 1200×1200, and for the Interval FUDJ, the number of buckets which is used to divide the time span into equal segments is set to 1000. Finally, for Text-similarity FUDJ, we use 0.9 as our similarity threshold since the algorithm is an exact similarity algorithm and higher thresholds is useful when it comes to analysis of similar reviews that have different overall ratings. In this experiment, the Spatial FUDJ demonstrates a speedup of around 1200x, while the Text-similarity FUDJ achieves a 6.5x improvement, and the Interval FUDJ delivers approximately a 2.5x boost in performance. Since the on-top approach cannot scale for Text-similarity and Interval joins, these speed upds had to be measured for small datasets. Hence, the speed up compared to the Spatial FUDJ seem smaller. In addition, we observe a high correlation between the performance of Text-similarity join and the dataset characteristics. We further discuss this in the following sections. Finally, we also observe that Interval join suffers mostly from NLJ operator that handles the bucket matching. While FUDJ framework can utilize HJ for Text-similarity and Spatial joins, it has to use NLJ since its matching function is a theta function.

Another aspect of the experiment that shown in Figure 3.11 the overhead caused by the FUDJ extensible framework is minimal. The difference between FUDJ and Built-in methods for Spatial and Interval joins are approximately 0 per record, while it is 0.061 ms. for Text-similarity. This cost can be explained by the cost of having summaries and config object as Hash Maps.

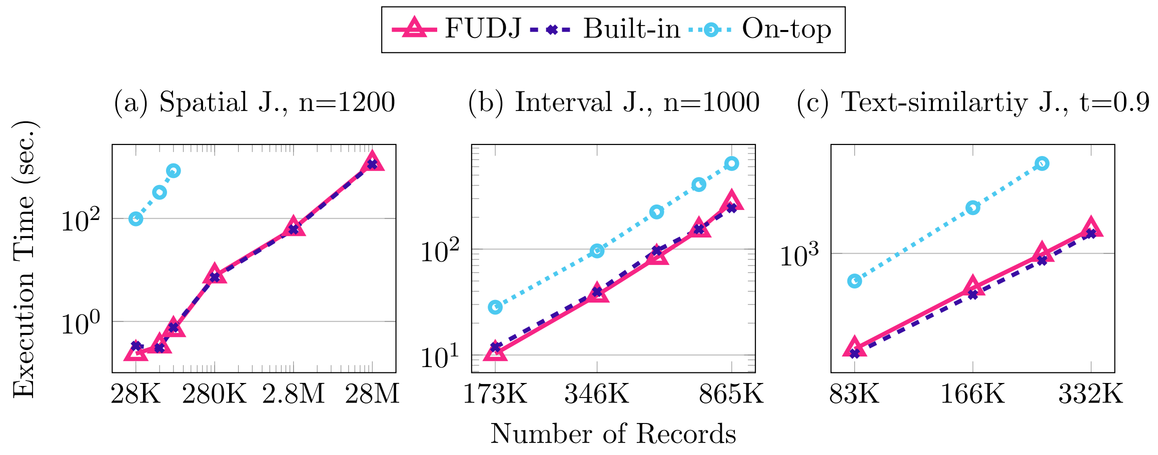


Figure 3.11: FUDJ, Built-in and On-top Query Execution Times of Spatial, Interval, and Text-similarity Join Examples For Various Dataset Sizes with 12 cores.

3.8.3 Scalibility

Here we evaluate the scalability of our design. We present query execution times of three versions of each algorithm by changing both the number of cores for joins and dataset sizes.

Figure 3.13 shows that Spatial and Text-similarity FUDJ algorithms scales well compare to the on-top approach. Besides that, it can be seen the difference between the built-in and FUDJ implementations remains limited as we increase the number of the cores and the data size. As a result, we can say that FUDJ is not causing any issues from the scalability perspective. On the other hand, As can be seen from the charts for Interval FUDJ, we cannot say the scaling is promising. This is due to the multi-join notion of the Interval FUDJ that result in NLJ operator utilized during the partition matching phase.

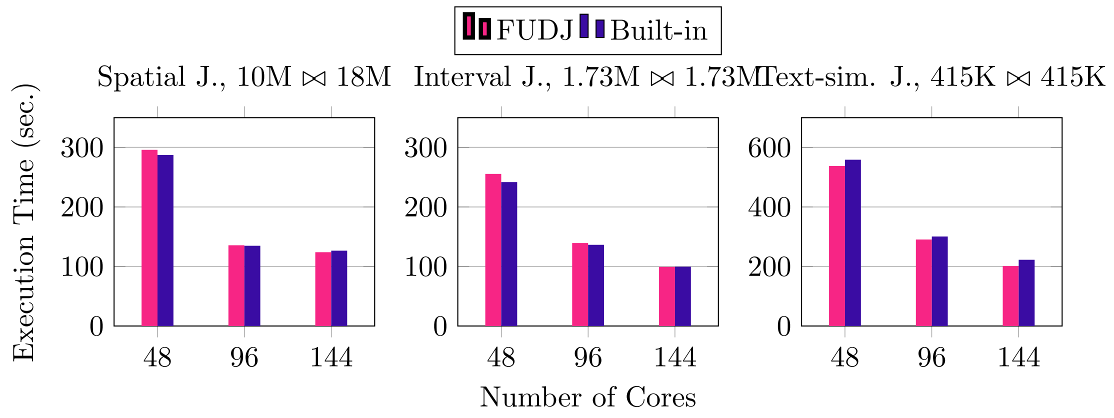


Figure 3.12: FUDJ Query Execution Times vs Dataset Size

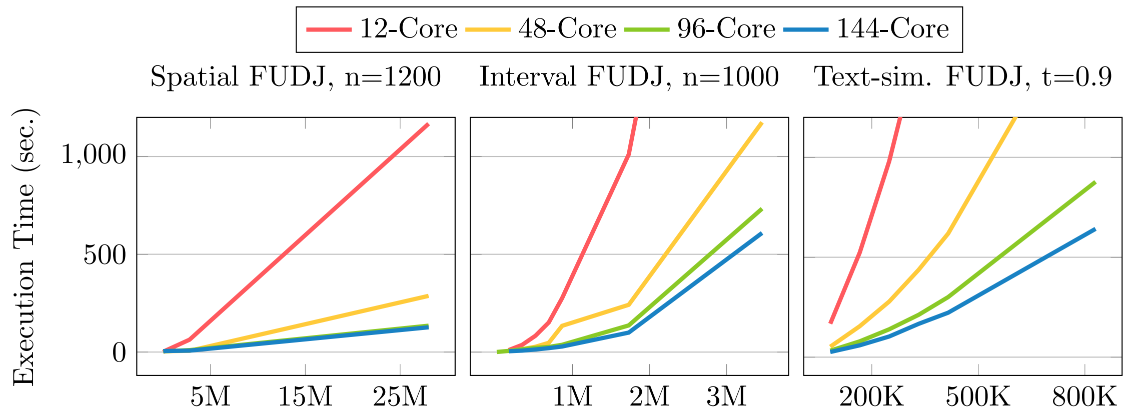


Figure 3.13: FUDJ Query Execution Times vs Dataset Size

Since there is not an efficient Theta Join operator available in Apache AsterixDB, this operation requires one side is randomly partitioned hence result in performance degrading. We are aware of this limitation of FUDJ and in the progress of development for an efficient Theta Join operator as future work.

3.8.4 Characteristics of the FUDJ Algorithms

In this section, we analyze the characteristics of the FUDJ algorithms and the datasets. First we study the effect of the number of buckets for Spatial FUDJ, and Interval FUDJ. Then, we show how similarity threshold effects the Text-similarity FUDJs performance.

Number of buckets

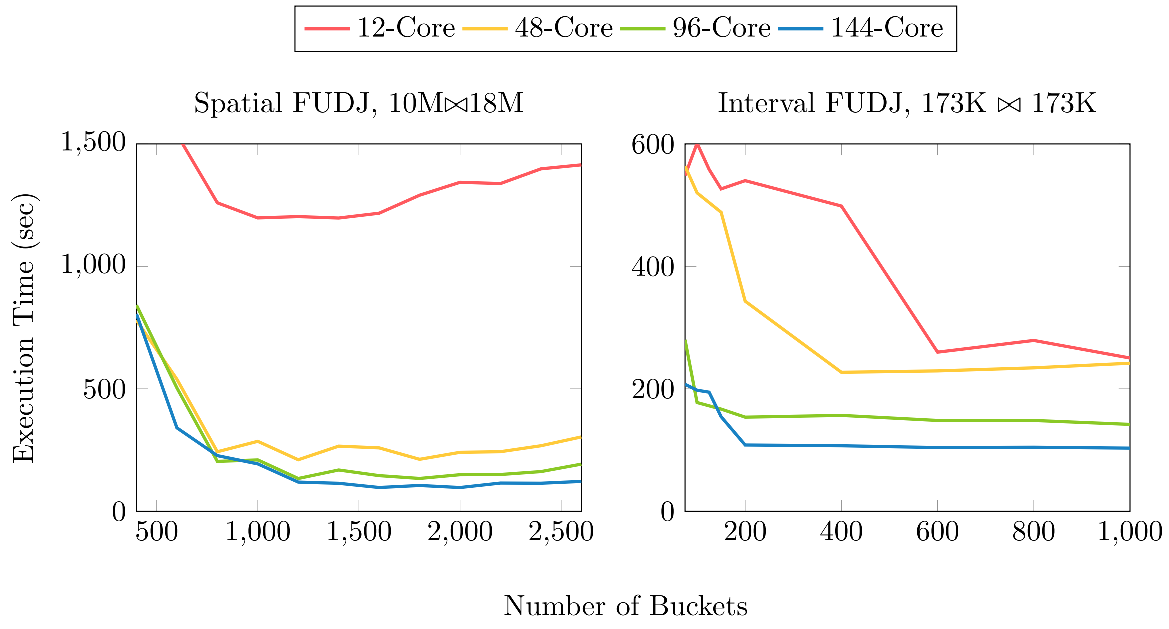


Figure 3.14: The Effect of Number of Buckets

The decision of the number of buckets is a crucial step for any distributed join algorithm. Before starting to evaluate FUDJ framework, we first analyze the logical characteristics of the FUDJ algorithms and dataset. As we discussed in Section 3.4, this step is crucial and a big challenge in complex join query processing.

For Spatial FUDJ and Interval FUDJ, we test the performance of the query processing by varying the number of buckets and measure the query execution times and show the results on Figure 3.14.

Similarity threshold

On the other hand, although Text similarity FUDJ does not require a number of bucket is determined, the characteristic of the dataset and most importantly the *similarity threshold* is the main factor for the execution performance. Furthermore, due to the duplication and prefix filtering method, it loses its benefits for thresholds above 0.6 in this scenario as can be seen from Figure 3.15.

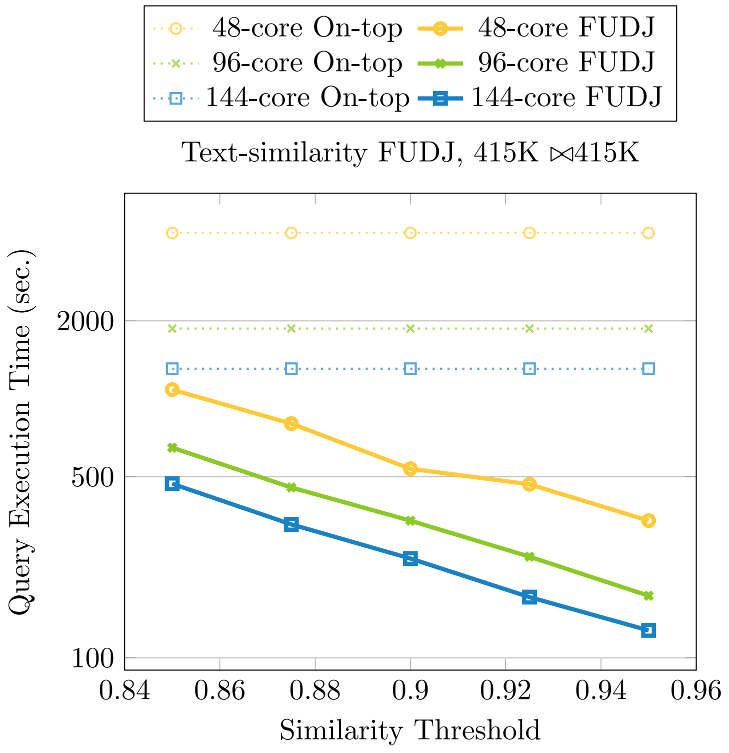


Figure 3.15: The Effect of Similarity Threshold

We used the best performing number of buckets for Spatial and Interval FUDJ experiments. For Text-similarity FUDJ, we pick 0.9 as the similarity threshold since the goal of the query is to find how 5 star reviews are similar to the 4 star reviews.

3.8.5 Duplicate Handling Methods

Duplicate handling is an important aspect of multi-assign optimized join algorithms as we discussed in Section 3.4. In FUDJ framework, the default duplicate handling method is Duplicate Avoidance since it is more promising by not requiring an additional shuffling stage after bucket matching. As a result the Text-similarity FUDJ is using the Duplicate Avoidance in contrast to the proposed method in its original study [53]. In this section, we first test the performance of these two methods on Text-similarity join. Figure 3.16 shows that Duplicate Avoidance outperforms Duplicate Elimination in all of the dataset sizes in the experiment by providing 1.15x speedup in average.

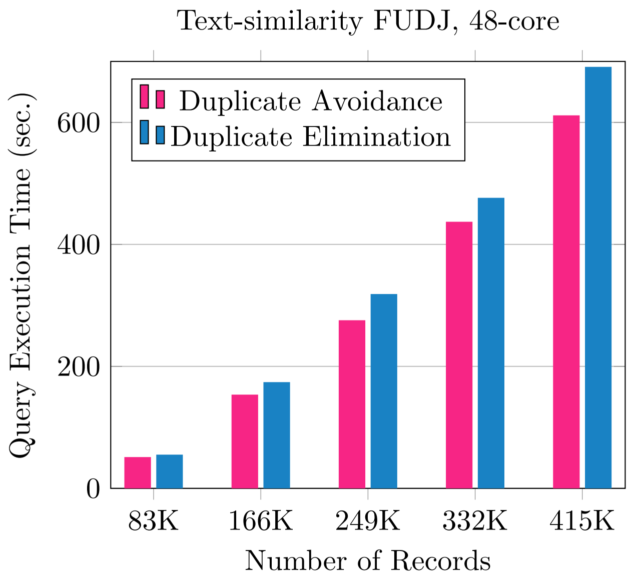


Figure 3.16: Text-similarity FUDJ Duplicate Handling: Duplicate Avoidance vs Elimination

The FUDJ programming model also allows the developers to implement their own Duplicate Avoidance methods. For instance, in Spatial FUDJ, we implement the Reference Tile method that described in [72] and compare the query execution performances of both methods for a various number of buckets. Since the number of buckets is the biggest factor in the duplication, we measure execution times for a variety of numbers. Figure 3.17 shows that there is not any dramatic difference in between the Reference Tile and FUDJ's duplicate avoidance methods. Consequently, we show that our default method can compete with one of the most successful Duplicate Avoidance methods without any tuning from the DBMS admin or implementation from the developer.

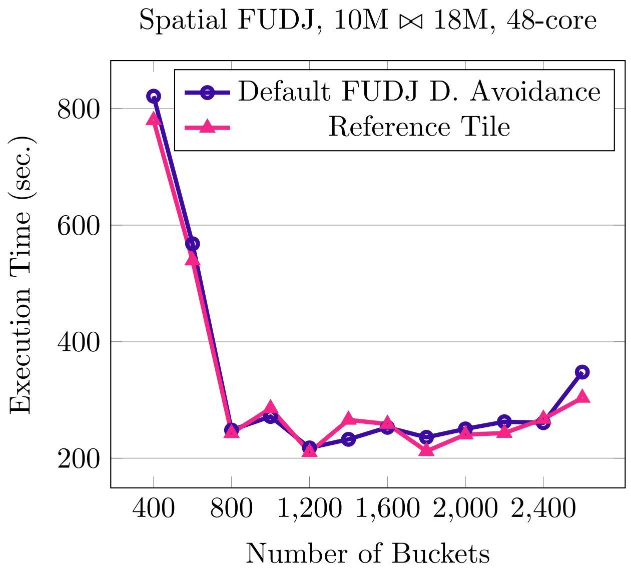


Figure 3.17: Spatial FUDJ Duplicate Handling: Default Method vs Reference Tile

3.8.6 Advanced Optimization Evaluations

Finally, we will discuss the performance improvement potential of FUDJ by comparing it to the implementations that involves advanced optimization techniques. For that purpose, we implement the PBSM [72] algorithm on Apache AsterixDB with a highly customized Spatial Join Operator. This operator's main advantage compare to the FUDJ version is that it can apply local optimizations while joining the buckets. In detail, it applies plane-sweep method that described in [72] by first sorting the geometries in each tile and applying the spatial-merging to efficiently joining geometries within each tile. Figure 3.18 shows that having local optimization for spatial joins yields 1.38x speedup in average.

Please note that with the current join operators in Apache AsterixDB, allowing developers to implement their custom local joining mechanism is not possible. We will address this limitation in the following research work and propose new operators that can handle this optimization.

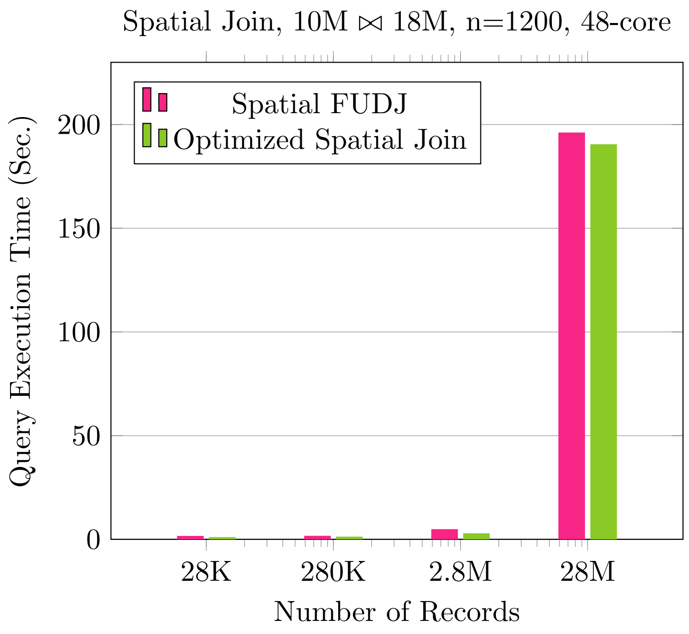


Figure 3.18: FUDJ Spatial Join vs Optimized Spatial Join

3.9 Conclusions and Future Work

By offering FUDJ, we can revolutionize the way join algorithms are implemented in data analysis. Such a system would empower users with varying levels of expertise to efficiently leverage join algorithms, significantly reducing the code and knowledge required for implementation. The utilization of native data types, flexible query execution plans, integration with the query optimization engine, easy installation of compact join libraries, and comparable performance to built-in implementations would unlock new possibilities for efficient join operations. Ultimately, this system would facilitate more comprehensive data analysis, uncover hidden insights, and drive accurate decision-making in diverse industries and applications.

3.9.1 Future Work

Number of buckets

Finding the optimum number of buckets is vital but complicated at the same time for the methods such as Spatial and Interval FUDJ. Our plan is to automize this decision by taking advantage of having the SUMMARIZATION step in which at the same time we can collect statistics about the dataset with minimum overhead. Figure 3.14 shows a pattern in the query time as we vary number of buckets. Next version of FUDJ framework shall utilize sampling or machine learning based methods to determine the number of buckets seamlessly by relying on these patterns.

Theta and Ternary Join Operators

Similar to the other DBMSs, in the current design of Apache AsterixDB, the join operators designed as to take two inputs and produce an output by applying a condition to them. Also, out of the box there are is not a Theta Join operator that we could utilize to process multi-joins or , and there is not a Ternary Join Operator that we could utilize to merge *match* and $verify$ operations hence we would not need to transfer the data in frames from matching join operator to the verifying join operator.

Chapter 4

Conclusions

In conclusion, our research has introduced two innovative solutions, HQ-Filter and FUDJ, each addressing critical challenges in the realm of data exploration and analysis.

HQ-Filter, designed for visual data exploration systems, effectively utilize the inherent hierarchy of data to enhance filtering capabilities across all levels of visualization in the user interface. This approach optimizes query performance, enabling the handling of large datasets, reducing response times, and alleviating server workloads. Our experiments have demonstrated its efficiency, and how it enhances the overall user experience in data exploration systems.

On the other hand, FUDJ presents a new implementation approach for the join algorithms to increase their availability by being a democratizing their usage across a spectrum of user expertise levels. By seamlessly integrating with query optimization engines, supporting native data types, and offering flexible execution plans, FUDJ streamlines the implementation of join operations. This advancement not only simplifies the code required but also opens up new possibilities for efficient join operations, ultimately leading to more comprehensive data analysis, revealing hidden insights, and empowering accurate decision-making across diverse industries and applications.

Together, HQ-Filter and FUDJ represent significant contributions to the field of data exploration and analysis, addressing complex challenges and enabling more efficient and accessible data processing for a wide range of users and scenarios.

Bibliography

[1] Foto N. Afrati, Anish Das Sarma, David Menestrina, Aditya G. Parameswaran, and Jeffrey D. Ullman. Fuzzy joins using mapreduce. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 498–509. IEEE Computer Society, 2012.

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

[3] Sattam Alsubaiee et al. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[4] Petko Bakalov, Marios Hadjieleftheriou, Eamonn Keogh, and Vassilis J Tsotras. Efficient trajectory joins using symbolic representations. In *Proceedings of the 6th international conference on Mobile data management*, pages 86–93, 2005.

[5] Petko Bakalov and Vassilis J Tsotras. Continuous spatiotemporal trajectory joins. In *GeoSensor Networks: Second International Conference, GSN 2006, Boston, MA, USA, October 1-3, 2006, Revised Selected and Invited Papers*, pages 109–128. Springer, 2008.

[6] Maximilian Bandle, Jana Giceva, and Thomas Neumann. To partition, or not to partition, that is the join question in a real system. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 168–180. ACM, 2021.

[7] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, 2016.

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130, 2010.

[9] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 131–140, New York, NY, USA, 2007. Association for Computing Machinery.

[10] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSRTR2014*, 41, 2014.

[11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *TVCG*, 17(12):2301–2309, 2011.

[12] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proc. VLDB Endow.*, 10(11):1346–1357, 2017.

[13] Panagiotis Bouros and Nikos Mamoulis. Spatial joins: What's next? *SIGSPATIAL Special*, 11(1):13–21, aug 2019.

[14] Francesco Cafagna and Michael H. Böhlen. Disjoint interval partitioning. *VLDB J.*, 26(3):447–466, 2017.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[16] Michael Carey and Laura Haas. Extensible database management systems. *ACM SIGMOD Record*, 19(4):54–60, 1990.

[17] Jr. Carman, Eldon P. *Interval Joins for Big Data*. PhD thesis, 2020. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-06-21.

[18] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing interval joins on map-reduce. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, pages 463–474. OpenProceedings.org, 2014.

[19] Lisi Chen, Shuo Shang, Christian S. Jensen, Bin Yao, and Panos Kalnis. Parallel semantic trajectory similarity join. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 997–1008. IEEE, 2020.

[20] Yun Chen and Jignesh M Patel. Design and evaluation of trajectory join algorithms. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 266–275, 2009.

[21] Andrew Crotty et al. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2027, 2015.

[22] Manoranjan Dash et al. An interactive analytics tool for understanding location semantics and mobility of users using mobile network data. In *MDM*, 2014.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[25] Ibrahim Dellal et al. On addressing the empty answer problem in uncertain knowledge bases. In *DEXA*, 2017.

[26] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 340–351. IEEE Computer Society, 2014.

[27] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. An efficient partition based method for exact set similarity joins. *Proc. VLDB Endow.*, 9(4):360–371, 2015.

[28] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1459–1470. ACM, 2014.

[29] Xin Ding et al. VIPTRA: visualization and interactive processing on big trajectory data. In *MDM*, 2018.

[30] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Idebench: A benchmark for interactive data exploration. In *SIGMOD*, 2020.

[31] Ahmed Eldawy and Mohamed F. Mokbel. Boundaries of parks and green areas from all over the world as extracted from openstreetmap., 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?OSM2015/parks&d`.

[32] Ahmed Eldawy, Mohamed F. Mokbel, and Christopher Jonathan. HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data. In *ICDE*, 2016.

[33] Sneha Gathani, Peter Lim, and Leilani Battle. Debugging database queries: A survey of tools, techniques, and users. In *CHI*, 2020.

[34] Saheli Ghosh and Ahmed Eldawy. AID*: A Spatial Index for Visual Exploration of Geo-Spatial Data. 2020.

[35] Saheli Ghosh, Ahmed Eldawy, and Shipra Jais. AID: an adaptive image data index for interactive multilevel visualization. In *ICDE*, 2019.

[36] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special*, 11(2):34–40, December 2019.

[37] Hua Guo et al. A case study using visualization interaction logs and insight metrics to understand how analysts arrive at insights. *TVCG*, 22(1):51–60, 2016.

[38] Himanshu Gupta, Bhupesh Chawda, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing multi-way spatial joins on map-reduce. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 113–124. ACM, 2013.

[39] Ruining He and Julian J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao, editors, *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517. ACM, 2016.

[40] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[41] Abdeltawab M. Hendawi et al. An interactive map-based system for visually exploring and cleaning GPS traces. In *SIGSPATIAL*. ACM, 2019.

[42] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, Genoa, Italy, March 2013. ACM.

[43] Thomas Hütter, Nikolaus Augsten, Christoph M. Kirsch, Michael J. Carey, and Chen Li. JEDI: these aren't the JSON documents you're looking for? In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1584–1597. ACM, 2022.

[44] IDC. Revenue from big data and business analytics worldwide from 2015 to 2022 (in billion u.s. dollars, April 2019. last visited March 11, 2020.

[45] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, 2015.

[46] Edwin H. Jacox and Hanan Samet. Iterative spatial join. *ACM Trans. Database Syst.*, 28(3):230–256, sep 2003.

[47] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7–es, mar 2007.

[48] Jianfeng Jia, Chen Li, and Michael J. Carey. Drum: A rhythmic approach to interactive analytics on large data. In *IEEE BigData*, pages 636–645, Boston, MA, December 2017.

[49] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Ajoin: Ad-hoc stream joins at scale. *Proc. VLDB Endow.*, 13(4):435–448, dec 2019.

[50] Nikolai Karpov and Qin Zhang. Syncsignature: A simple, efficient, parallelizable framework for tree similarity joins. *Proc. VLDB Endow.*, 16(2):330–342, oct 2022.

[51] Martin L. Kersten et al. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.

[52] Taewoo Kim, Wenhai Li, Alexander Behm, Inci Cetindil, Rares Vernica, Vinayak Borkar, Michael J Carey, and Chen Li. Similarity query support in big data management systems. *Information Systems*, 88:101455, 2020.

[53] Taewoo Kim, Wenhai Li, Alexander Behm, Inci Cetindil, Rares Vernica, Vinayak R. Borkar, Michael J. Carey, and Chen Li. Similarity query support in big data management systems. *Inf. Syst.*, 88, 2020.

[54] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, September 2008.

[55] Kaiyu Li and Guoliang Li. Approximate query processing: What is new and where to go? - A survey on approximate query processing. *Data Sci. Eng.*, 3(4):379–397, 2018.

[56] Lauro Didier Lins, James T. Klosowski, and Carlos Eduardo Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *TVCG*, 19(12):2456–2465.

[57] Jonathan Liono et al. UTE: A ubiquitous data exploration platform for mobile sensing experiments. In *MDM*, 2016.

[58] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *TVCG*, 20(12):2122–2131, 2014.

[59] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.

[60] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. *SIGMOD Rec.*, 25(2):247–258, jun 1996.

[61] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 247–258, New York, NY, USA, 1996. Association for Computing Machinery.

[62] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 5(10):1016–1027, jun 2012.

[63] Gang Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, Seoul, Korea, September 2006. ACM.

[64] Xinze Lyu and Wei Hu. RQE: rule-driven query expansion to solve empty answers in SPARQL. In *JIST*, 2019.

[65] Willi Mann and Nikolaus Augsten. PEL: position-enhanced length filter for set similarity joins. In Friederike Klan, Günther Specht, and Hans Gamper, editors, *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014*, volume 1313 of *CEUR Workshop Proceedings*, pages 89–94. CEUR-WS.org, 2014.

[66] Davide Mottin et al. IQR: an interactive query relaxation system for the empty-answer problem. In *SIGMOD*, pages 1095–1098, Snowbird, UT, June 2014. ACM.

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

[68] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *PVLDB*, 6(14):1762–1773, 2013.

[69] Sadegh Nobari, Farhan Tauheed, Thomas Heinis, Panagiotis Karras, Stéphane Bressan, and Anastasia Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 701–712, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 949–960, New York, NY, USA, 2011. Association for Computing Machinery.

[71] Yongjoo Park, Michael J. Cafarella, and Barzan Mozafari. Visualization-aware sampling for very large databases. In *ICDE*, 2016.

[72] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 259–270. ACM Press, 1996.

[73] Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62–78, 2011. Selected Papers from the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009).

[74] Julian Ramos Rojas et al. Sampling techniques to improve big data exploration. In *LDAV*, 2017.

[75] Mohamed Sarwat. Interactive and scalable exploration of big spatial data - A data management perspective. In *MDM*, 2015.

[76] Abdullah Sawas et al. Trajectolizer: Interactive analysis and exploration of trajectory group dynamics. In *MDM*, 2018.

[77] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1961–1976, New York, NY, USA, 2016. Association for Computing Machinery.

[78] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the starburst database system. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems*, OODS '86, page 85–92, Washington, DC, USA, 1986. IEEE Computer Society Press.

[79] Amirhesam Shahvarani and Hans-Arno Jacobsen. Distributed stream KNN join. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1597–1609. ACM, 2021.

[80] Salman Ahmed Shaikh, Komal Mariam, Hiroyuki Kitagawa, and Kyoung-Sook Kim. Geoflink: A distributed and scalable framework for the real-time processing of spatial streams. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 3149–3156, New York, NY, USA, 2020. Association for Computing Machinery.

[81] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S Jensen, Kai Zheng, and Panos Kalnis. Parallel trajectory similarity joins in spatial networks. *The VLDB Journal*, 27(3):395–420, 2018.

[82] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. DITA: distributed in-memory trajectory analytics. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 725–740. ACM, 2018.

[83] A. B. Siddique, Ahmed Eldawy, and Vagelis Hristidis. Comparing synopsis techniques for approximate spatial data analysis. *PVLDB*, 12(11):1583–1596, 2019.

[84] Yasin N. Silva, Walid G. Aref, and Mohamed H. Ali. The similarity join database operator. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 892–903, 2010.

[85] Yasin N. Silva, Spencer S. Pearson, Jaime Chon, and Ryan Roberts. Similarity joins: Their implementation and interactions with other database operators. *Information Systems*, 52:149–162, 2015. Special Issue on Selected Papers from SISAP 2013.

[86] S. Singla, T. Diao, A. Mukhopadhyay, A. Eldawy, R. Shachter, and M. Kochenderfer. Wildfiredb : an open-source dataset that links wildfire occurrence with relevant features, 2021. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?wildfiredb&d`.

[87] Chris Stolte and Pat Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In *INFOVIS*, pages 5–14, Sald Lake City, October 2000. IEEE Computer Society.

[88] Simon Su et al. Visually analyzing A billion tweets: An application for collaborative visual analytics on large high-resolution display. In *IEEE BigData*, pages 3597–3606, Seattle, WA, December 2018. IEEE.

[89] Na Ta, Guoliang Li, Yongqing Xie, Changqi Li, Shuang Hao, and Jianhua Feng. Signature-based trajectory similarity join. *IEEE Trans. Knowl. Data Eng.*, 29(4):870–883, 2017.

[90] Usgs earthquake map, 2020.

[91] Elena Vasilyeva et al. Answering ”why empty?” and ”why so many?” queries in graph databases. *J. Comput. Syst. Sci.*, 82(1):3–22, 2016.

[92] Elena Vasilyeva, Thomas Heinze, Maik Thiele, and Wolfgang Lehner. Debeaq - debugging empty-answer queries on large data graphs. In *ICDE*, pages 1402–1405, Helsinki, Finland, May 2016.

[93] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 495–506, New York, NY, USA, 2010. Association for Computing Machinery.

[94] Tin Vu, Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. Towards a learned cost model for distributed spatial join: Data, code & models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 4550–4554, New York, NY, USA, 2022. Association for Computing Machinery.

[95] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 85–96. ACM, 2012.

[96] Meng Wang et al. Towards empty answers in SPARQL: approximating querying with RDF embedding. In *ISWC*, 2018.

[97] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, and Gao Cong. A survey on trajectory data management, analytics, and learning. *ACM Comput. Surv.*, 54(2), mar 2021.

[98] Martin Werner. Globimaps - A probabilistic data structure for in-memory processing of global raster datasets. In *SIGSPATIAL*, pages 3–12, Chicago, IL, November 2019. ACM.

[99] Richard Michael Grantham Wesley, Matthew Eldridge, and Pawel Terlecki. An analytic data engine for visualization in tableau. In *SIGMOD*, 2011.

[100] Leland Wilkinson. *The Grammar of Graphics, Second Edition.* Statistics and computing. Springer, 2005.

[101] Chris Wong. Pickup and drop-off locations of taxi rides in new york city, 2019. Retrieved from UCR-STAR `https://star.cs.ucr.edu/?NYCTaxi&d`.

[102] Kanit Wongsuphasawat et al. Towards a general-purpose query language for visualization recommendation. In *HILDA*, 2016.

[103] Kanit Wongsuphasawat et al. Voyager 2: Augmenting visual analysis with partial view specifications. In *CHI*, pages 2648–2659. ACM, 2017.

[104] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. Mobility and extensibility in the stratosphere framework. *Distributed Parallel Databases*, 7(3):289–317, 1999.

[105] Eugene Wu, Leilani Battle, and Samuel Madden. The case for data visualization management systems. *PVLDB*, 7(10):903–906, 2014.

[106] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker Jr. Map-reduce-merge: simplified relational data processing on large clusters. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1029–1040. ACM, 2007.

[107] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 34–41, 2015.

[108] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 34–41, 2015.

[109] Jia Yu and Mohamed Sarwat. Turbocharging geospatial visualization dashboards via a materialized sampling cube approach. In *ICDE*, 2020.

[110] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem. In *SSDBM*, 2018.

[111] Haitao Yuan and Guoliang Li. Distributed in-memory trajectory similarity search and join on road network. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1262–1273. IEEE, 2019.

[112] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[113] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[114] Hao Zhang, Miao Qiao, Jeffrey Xu Yu, and Hong Cheng. Fast distributed complex join processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 2087–2092. IEEE, 2021.

[115] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: parallelizing spatial join with mapreduce on clusters. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–8. IEEE Computer Society, 2009.

[116] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[117] Xiaofang Zhou, David J Abel, and David Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2:175–204, 1998.