FUDJ: Flexible User-Defined Distributed Joins

Akil Sevim* Ahmed Eldawy* E. Preston Carman Jr.† Michael J. Carey§ Vassilis J. Tsotras*
*University of California, Riverside
†Walla Walla University, Washington
§University of California, Irvine
{asevi006, eldawy, vtsotras}@ucr.edu preston.carman@wallawalla.edu mjcarey@ics.uci.edu

***Abstract*—Join operations are crucial in data analysis, but can suffer inefficiency with large datasets and complex non-equality-based conditions. Optimized join algorithms have gained traction in database research to address these challenges. One popular choice for implementing join algorithms is distributed data processing frameworks, e.g., Hadoop and Spark, but each implementation is highly tailored for specific query types. As a result, they do not address join queries that involve diverse and complex conditions since they are not integrated into a holistic query optimization engine like in DBMSs. On the other hand, implementing new join algorithms on a DBMS from scratch requires substantial effort and expertise. This paper introduces FUDJ, Flexible User-defined Distributed Joins, a framework for complex distributed join algorithms. The key idea of FUDJ is to allow developers to realize new distributed join algorithms into the database without delving into the database internals. As shown, an algorithm implemented in FUDJ is up to an order of magnitude faster than existing user-defined implementations with an order of magnitude fewer lines of code.**

***Index Terms*—distributed joins, database extensibility**

I. Introduction

Joining datasets is a fundamental task that has been extensively studied for decades. Historically, Database Management Systems (DBMS) treated "join" as an operation for structured data with simple conditions like equality. However, with the growing volume and variety of data and the rise of data-driven applications, various other types of join operations are becoming increasingly popular. Today, data scientists must combine large and diverse datasets from sources like social networks and IoT devices using distributed systems. This calls for optimized and complex join queries that operate on diverse data types. As a result, there has been significant research in the area. However, the availability of optimization techniques for the new join types in DBMSs remains limited due to implementation and integration complexities as explained below.

Currently, there are four methods for implementing new join operators. First is the **on-top** approach that implements the join predicate as a user-defined function (UDF) which the DBMS uses with nested-loop join (NLJ). While easy to implement, this approach has a limited performance due to the cost of the nested loop. Second is the **standalone** approach [1]–[6], where developers independently craft algorithms without any platform integration. Third is the use of the programming paradigm of a **distributed system**, such as Spark [7], [8], Hadoop [9]–[12], or Flink [13], [14]. The last two approaches can be highly optimized but have limited application since they cannot be integrated into a DBMS directly where users want to perform all their analyses with the support of a comprehensive query optimizer.

Besides these methods, a few studies have proposed a fourth approach to implement a **built-in** optimized join within a full-fledged DBMS, e.g., set similarity join on PostgreSQL [15] and AsterixDB [16], interval join [17] on AsterixDB, and spatial join on Paradise [18]. 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. 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.

A. Motivation

To better clarify the importance of complex join query optimization, 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 DATASET Parks(Park_Type) PRIMARY KEY id;
CREATE TYPE Wildfire_Type {id: uuid, lat: float , lon: float ,
        fire_start : datetime , fire_end : datetime };
CREATE DATASET Wildfires(Wildfire_Type) PRIMARY KEY id;
```

Type 1: Parks and Wildfires Type Definitions

To find recently damaged parks, the scientist wants to run the spatial join query shown in Query 1 with the computationally expensive predicate *ST_Contains* that detects whether a wildfire location is contained by another park boundary polygon. Note that Query 1 is not only a join query but involves other operations like filtering, aggregation, and sorting.

```
SELECT p.id, p.tags, p.boundary, COUNT(w.id) AS num_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 1: Spatial Join Query

This research was supported in part by NSF award IIS-1838222, CNS-1924694, IIS-1954962, IIS-1924694, IIS-1954644, IIS-2046236 and by the Donald Bren Foundation (via a Bren Chair.)

Despite all existing works for spatial joins, finding a distributed big data processing system with an efficient execution plan for Query 1 is rare. The Hash Join (HJ) algorithm is unsuitable due to its equi-join requirement, limiting options to the NLJ operator. An alternative is leveraging a spatial index with the Indexed-Nested Loop Join (INLJ) operator. However, INLJ works well only when the non-indexed set is relatively small. So, challenges persist in scalability and resource utilization for large datasets. After Query 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 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". Assume that damaged parks were stored in "Damaged_Parks" dataset after Query 1.

```
SELECT dp.park_id, p.id,  jaccard_similarity (dp.tags, p.tags) as sim
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, sim;
```

Query 2: Text-similarity Join

Next, the team may want to investigate the relationship between the weather and wildfires by using the "Weather" dataset with the schema shown in Type 2.

```
CREATE TYPE Weather_Type {id: uuid, location: geometry,
      reading_interval: interval, temp: int};
CREATE DATASET Weather(Weather_Type) PRIMARY KEY id;
```

Type 2: Weather Type Definition

To find the average temperature for each wildfire that has happened in each park, they can use a combination of spatial and interval joins: Query 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.

Both Query 2 and Query 3 would likely end up being 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 specialized tools for spatial join queries for Query 1. Further, note that Query 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.

```
SELECT f.id, f. fire_start , AVG(w.temp)
FROM Wildfires f, Parks p, Weather 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: Interval and Spatial Join Query

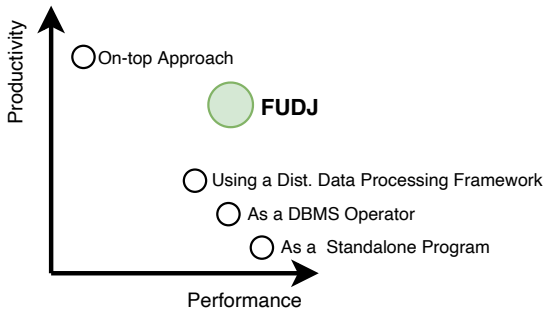


Fig. 1: Productivity and Performance of Existing Optimized Join Implementation Methods

B. 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, those optimized algorithms would efficiently process the queries above, enabling faster data analysis. This work introduces the *Flexible User-defined Distributed Joins* (FUDJ) framework, which enhances the availability of optimized join algorithms within DBMSs. FUDJ allows users to implement 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 1 illustrates a high-level summary of the performance and productivity evaluation of the current implementation methods and NLJ (on-top approach) for the optimized join algorithms. FUDJ's position shows it aims to provide high productivity and maintain on-par performance compared to other options.

To achieve these goals, we propose a novel extensibility architecture for implementing new join algorithms into a distributed 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 similar in spirit to User-Defined Aggregates (UDAs) [19], where users provide a function that aggregates a large set of values by computing partial aggregates on partitions and then combining them to compute the final result. FUDJ allows 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) [20]. In GiST, the common logic, such as node merging and splitting, is implemented in the code base of the DBMS while the developer defines index-specific logic, such as comparison operations within tree nodes. In FUDJ, 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:** A new programming model that allows developers to implement existing or new partition-based distributed join algorithms without having database internal and distributed programming experience.
- **FUDJ Infrastructure:** Design of components to support FUDJ that could be applied to any DBMS with the following generic extensions:
  - Install join libraries (e.g., 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 functions in the FUDJ library.
- Realization of the concept on AsterixDB as proof of its feasibility, and providing FUDJ implementations for Spatial, Overlapping Interval, and Set-Similarity Distributed Joins.
- Run extensive experiments showing that the FUDJ implementations require roughly 10x less work while providing as much as two orders of magnitude speed-up against on-top approaches, which is close to built-in approaches.

The rest of this paper is organized as follows. Section II discusses related work. Section III addresses the commonalities and challenges of distributed optimized join algorithms. Section IV presents our programming model, and Section V shares the details of the realization of the architecture on AsterixDB and describes three example join algorithm implementations. Section VI provides details about our framework and its application to query optimizers. Section VII-B explores the current performance of FUDJ, and Section VIII concludes our study and discusses possible future work.

II. Related Work

Both academia and industry have extensively studied **joins** in various domains. For instance, many studies propose methods for spatial joins [18], [21]–[24], while survey papers like [25]–[28] offer comprehensive evaluations of existing spatial join methods. Set-similarity joins have been considered in [9], [11], [12], [16], [29]–[33]. Trajectory joins are explored in [7], [8], [34]–[37]; and surveyed in [38]. JSON similarity studies have been addressed in [5], [6]. Interval joins have been examined in [1], [4], [17], [39], while kNN joins are explored in [40], [41]. It is important to note that each study introduces a method tailored for a specific join type. However, despite this rich literature, there is a scarcity of DBMSs that comprehensively support a diverse collection of join variations.

The typical **join implementation methods** can be classified into three categories: distributed data processing framework-based, as standalone programs, or as special DBMS operators. The implementations based on distributed data processing frameworks follow programming paradigms such as MapReduce [42], RDD [43], or PACT [44] depending on the system. Standalone implementations [1]–[6] usually build their systems from scratch. However, these approaches assume that join is a standalone program and ignore the realistic scenario where it is a part of a complex query plan. A select few approaches [16]–[18], [33], [39] 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.

Related to the concept of **database extensibility** [45] are commonly adopted concepts such as User-Defined Functions (UDFs) and User-Defined Aggregates (UDAs). The Generalized Search Tree (GiST) [20] introduces an extensible framework that allows developers to implement and integrate custom indexing methods. While GiSTs can enhance join performance in specific cases when used with Indexed Nested Loop joins, they lack the capability to integrate new join algorithms into a Database Management System (DBMS). As a result, the concept of database extensibility has not yet encompassed 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.

III. Common Challenges in Distributed Join

The strategies employed in optimized distributed join methods are crucial for scalable data analysis. In this context, three primary optimized join approach categories stand out: nested-loop joins, partition-based joins, and sort-merge-based joins [46]. Nested-loop joins follow a straightforward implementation to distribute the data but they exhibit limited optimization potential due to their brute-force nature.Sort-merge joins are preferred when the data is already sorted and are effective in parallelization for some cases. However, they encounter challenges in shared-nothing environments due to the need for data shuffling across nodes and sorting 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 the partition-based methodology the most popular choice for optimizing joins in distributed systems in numerous studies for various domains.

Since we aim to increase the availability of optimized joins in DBMSs, the FUDJ programming model that we introduce here 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

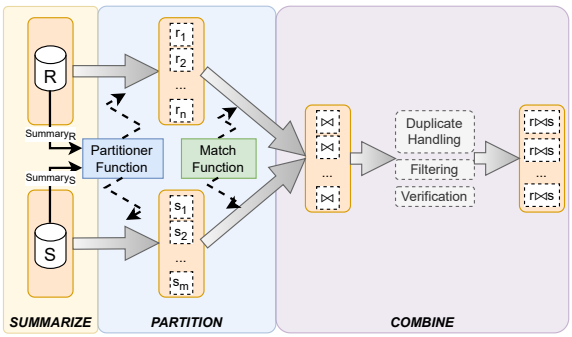
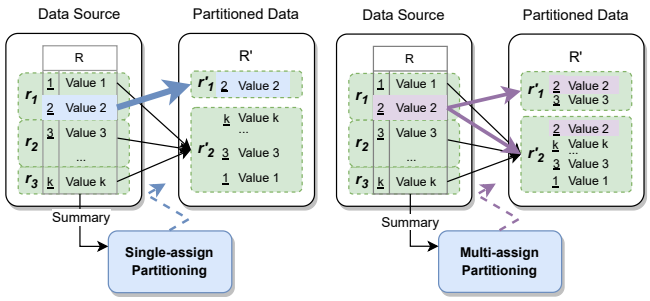


Fig. 2: Phases of Partition-based Distributed Joins



(a) Single-assign (b) Multi-assign

Fig. 3: Partitioning Methods

join techniques and injecting them into the code base of DBMSs while externalizing the logic related to specific join operations through user-defined joins that are implemented using the FUDJ programming model.

In the rest of this section, we identify the common challenges and features in the two main phases of partition-based joins, namely, *partition* and *join*, as shown in Figure 2.

A. Partitioning

The partitioning phase presents several challenges that require careful consideration [47]. One of the foremost challenges is achieving optimal data distribution across nodes. 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 cost of inter-node communication during subsequent join phases. Balancing partition granularity and size is yet another challenge. Overly fine-grained partitions might lead to excessive overhead caused by duplication, 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 Figure 2, an initial scan of the input dataset to collect such information (Summary) to have a better partitioning is one of the most common approaches. For instance, the OIPJoin algorithm [1] requires minimum interval start and maximum interval end times to divide the space into equal-sized granules. PBSM [18] computes the Minimum Bounding Rectangles (MBR) of the input and divides it into tiles. Finally, text-similarity join [48] counts the words from input datasets and sorts them by their occurrences to find the least common words in each record. In all these scenarios, the input space is then divided into **buckets** at the logical level, and each record is assigned to a physical partition accordingly by relying on buckets.

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 and deduplication. For instance, PBSM [18] assigns each geometry to all of the tiles that they are overlapping with. In text-similarity join [48], tokens within each text are sorted by rank based on their occurrence frequencies and then assigned to a specified number of least common words, determined by a 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 [1] assigns intervals to the smallest interval bucket that it can fit. Figure 3a illustrates the single-assign method since each record is assigned to only one partition after partitioning. On the other hand in Figure 3b, *Value 2* from partition r_1 is duplicated and assigned to both partition r'_1 and r'_2 which makes that partitioning method a multi-assign one.

In the rest of this paper, we use the terms that are defined below to refer to the partitioning phase elements.

Definition 1: Summarization: The phase where the join algorithms collect information about the data.

Definition 2: Summary: The data structure is where the information is aggregated during the *Summarization* phase.

Definition 3: Divide: The function that combines the *Summary* from both sides of the join and any other required information needed to determine the partitioning.

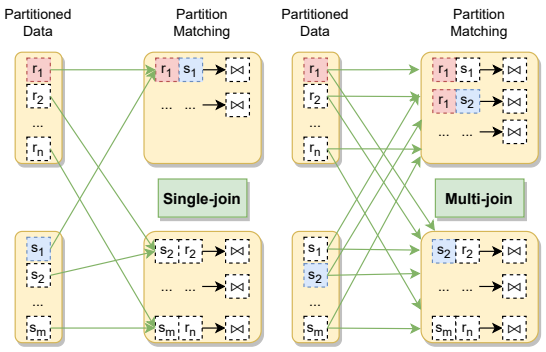
Definition 4: Partitioning Plan (PPlan): The data structure that holds the partitioning information returned by *Divide*.

Definition 5: Bucket: A group of records that are grouped based on the *Partitioning Plan* in a way that when the buckets are joined, the records in the buckets from both sides are potentially in the join result.

Definition 6: Assign: A function that determines which record should be in which *Bucket* based on the information provided by the*Partitioning Plan*.

B. Joining

One of the primary challenges in the joining phase is the task of joining the buckets. Eliminating irrelevant buckets from consideration or moving the buckets to the same nodes before the join operation can reduce unnecessary data movement and processing. The matching method for the buckets plays a crucial role in having an efficient strategy for efficient



(a) Single-join **(b) Multi-join**

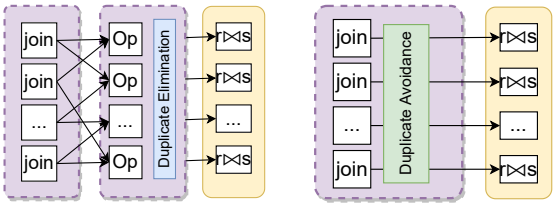
Fig. 4: Partition Matching Strategies

bucket joining. As shown in Figure 4, joining buckets can be categorized as single-join or multi-join. In **single-join**, each bucket on one side matches with a single bucket on the other side resulting in a one-to-one correspondence between buckets which can be efficiently done with hash-based join. For instance, PBSM [18] only joins the records that overlap with the same tiles, and in Set-similarity join [48], the records that share the same tokens are matched. In **multi-join**, a bucket can match with more than one bucket which makes it a theta-join operation. As a result, buckets from one side are broadcast in most of the cases. OIPJoin [1] is an example of that since one interval bucket can match with multiple buckets.

Local optimization strategies are also applied during bucket joins on each node. This includes implementing customized join algorithms within individual nodes to minimize computational and memory overhead. 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 of tuples within partitions to apply merge join algorithms which can reduce memory footprint.

In addition, partitioning strategies that involve duplicating tuples across multiple partitions (multi-assigning) can introduce duplicate handling challenges. Duplicate elimination becomes essential in subsequent stages, and it involves identifying and eliminating duplicate tuples from the joined output as illustrated in Figure 5a. Avoidance techniques, on the other hand, aim to prevent duplicates during the join process itself by cleverly designing matching and partitioning strategies. Hence unlike duplicate elimination, it does not require an additional step after joining as illustrated in Figure 5b. After 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 method that is used to match the buckets as **match**. The logic of the match function defines whether the join is a single-join or multi-join. For instance, if the match is a simple equality, then the join becomes a single-join and the system



(a) Duplicate Elimination **(b) Duplicate Avoidance**

Fig. 5: Partitioning Categories

can utilize its optimized hash join operator. Lastly, the function that verifies the tuple pairs to finalize the join operation will be called **verify**. The verify function can also use the **PPlan** from the partitioning phase to determine whether the tuple pair belongs to the final output or not.

Definition 7: Match: A boolean function that determines whether two buckets should be joined or not.

Definition 8: Verify: A boolean function that determines whether two records from matched buckets should be in the final result or not.

IV. Programming Model

To address the common challenges in partition-based distributed optimized join algorithms that we describe in Section III, this section introduces the FUDJ programming model that consists of three phases namely, **SUMMARIZE**, **PARTITION**, and **COMBINE**. Figure 6 shows all the functions within each phase. The rest of this section provides more details about the phases and the functions.

A. SUMMARIZE

To successfully decide how to partition the datasets, many join algorithms apply an initial step that analyzes and summarizes the data to produce better partitioning in the second phase. The summary can be the minimum bounding rectangle for a spatial dataset [18], minimum starting and maximum ending time for an interval dataset [1], or word frequencies for text-similarity joins [9]. Since FUDJ is designed for distributed systems, it follows a common two-step aggregation method that first aggregates data locally within each node and then combines the results to compute the final aggregate. We provide two aggregate function interfaces as below.

$$local\_aggregate(T\,key, SUMMARY\langle T\rangle S) : SUMMARY\langle T\rangle$$
$$global\_aggregate(SUMMARY\langle T\rangle\, S_1, SUMMARY\langle T\rangle\, S_2) : SUMMARY\langle T\rangle$$

The $local\_aggregate$ function reads keys from the input dataset and updates a $SUMMARY$ object. Then all $SUMMARY$ objects are merged into global $SUMMARY$ objects by a $global\_aggregate$ function. Note that the framework allows to have two versions of local and global aggregate functions one for each side of the join since key types can be different. If both sides should be summarized in the same way, the user provides one implementation only. In the case of self-join, the framework will optimize the computation by

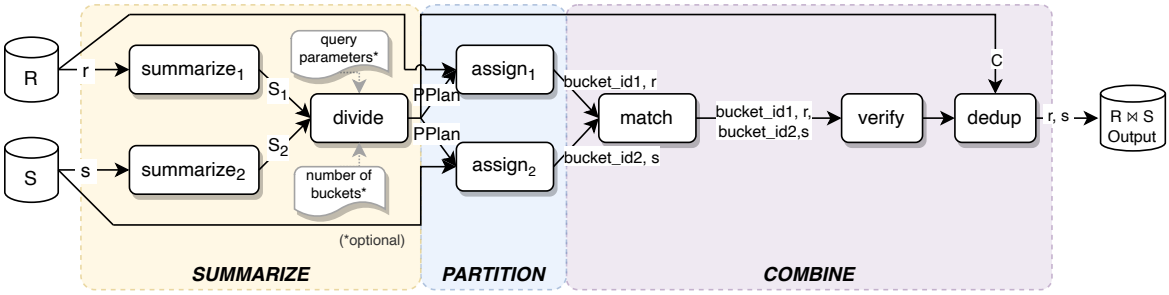


Fig. 6: Flexible User-Defined Distributed Join Data Flow Diagram

summarizing the data only once. For simplicity, function name $SUMMARIZE$ will be used to refer to $local_aggregate$ and $global_aggregate$ functions combined in the rest of the paper.

Lastly, to divide the input domain space into meaningful partitions, we provide the $DIVIDE$ function that takes two global $SUMMARY$ objects, one from each side of the join, and returns a $PPlan$ object.

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

For instance, in spatial join, $divide$ combines two MBRs from both sides and returns the grid information for the join. For interval join, it takes the minimum start time and maximum end time for both datasets and finds the number of interval partitions. For similarity join, it uses word counts from both sides to create the ordered list of the least common words.

B. PARTITION

The goal of this phase is to assign the input datasets into subgroups which we will call $buckets$, each identified with a unique integer $bucket\_id$. The framework will then use the list of $bucket\_id$'s and the logic of the join algorithm to decide how to partition the input datasets. For example, in a spatial join, a $bucket$ is a tile and $bucket\_id$ is the tile_id, while for text-similarity join, a $bucket$ is a word from the word count list and $bucket\_id$ is the rank of that word.

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

The partitioning phase scans the input datasets and applies the $assign$ function on each key to return a list of $bucket\_id$ which is computed based on $PPlan$. A key can be assigned to only one $bucket$ (Single-Assign) or multiple buckets (Multi-Assign). Similar to aggregate functions, a user can implement a different $assign$ function for each side of the join.

C. COMBINE

This final phase processes the data in all buckets to produce the final answer, i.e., pairs of matching records. First, we determine which $bucket$ matches with another $bucket$ by using the $match$ function. As mentioned before there are two cases in this stage: single-join or multi-join. For single-join algorithms, we provide a default $match$ function which checks whether both $bucket\_id$'s are the same or not. For this type of algorithm, the developer should just use the default implementation since further optimizations can be applied.

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

After buckets are matched, the next step is verifying the record pairs by using the *verify* function as below.

$$verify(T\,key1\,Tkey2) : boolean$$

As discussed in Section III, some algorithms yield duplication due to the assignment of records to multiple buckets. The user should implement the $dedup$ function as it handles the duplicates. FUDJ's default duplicate avoidance method relies on the utilization of the $assign$ functions with $PPlan$, and producing the list of $bucket_id$s for each record pair to find if the matching buckets are the first matching pair or not. For algorithms that do require a custom method for deduplication, the user can easily override the $dedup$ function provided by the framework, or it can be disabled if there is no need for the deduplication for more efficient query processing.

$$dedup(int\,bucket\_id1, T\,key1\ int\,bucket\_id2,\ Tkey2, PPlan\,C) : boolean$$

V. Example Implementations

This section provides the logic of three FUDJ example implementations for spatial, text-similarity, and overlapping interval joins. These examples represent the FUDJ versions of the algorithms that we discussed earlier in Section III. In the rest of the section $SUMMARY$ will be denoted by S.

A. Spatial FUDJ

Our Spatial FUDJ implementation is based on the PBSM algorithm described in [18]. We start by calculating the MBRs of each dataset with the $summarize$ function. Here, the $MBR()$ function returns the MBR of a given $geometry$ and the $\cup$ operator merges two MBRs and returns an MBR that covers both MBRs.

1: **function** SUMMARIZE($geometry$, S)
2: $\quad S \leftarrow MBR(geometry) \cup S$

After we compute MBRs from both sides of the join, we then use the $divide$ function to compute the final MBR and create the grid that divides the space into $n \times n$ buckets. Next, we store the final MBR and n into $PPlan$.

1: **function** DIVIDE(S_1, S_2, n)
2: $\quad MBR \leftarrow (S_1 \cap S_2)$
3: $\quad PPlan \leftarrow (MBR,\ n)$
4: $\quad$ **return** $PPlan$

Now, our spatial join algorithm can assign the data's geometries to relevant buckets. To simplify the algorithm here the function $getOverlappingTileIds()$ represents a function that logically divides the 2D space into $n \times n$ equal-sized tiles and returns the ids (numbered from 1 to $n^2 - 1$) of the tiles that overlap with the given geometry's MBR.

```
1: function ASSIGN(geometry, PPlan)
2:     MBR ← MBR(geometry)
3:     return getOverlappingTileIDs(MBR, PPlan)
```

When it comes to matching the data in buckets, since our algorithm follows the single-joining strategy, it can simply utilize the default equality-based $match$ function. Finally, we provide a simple $verify$ function that checks if the actual geometries are intersecting or not.

```
1: function VERIFY(tileId1, geometry1, tileId2, geometry2, PPlan)
2:     return intersects(geometry1, geometry2)
```

B. Text Similarity FUDJ

Similar to [16], we first count the words of all the records in the summary step by using a hash map that maps words to counts. Here the $tokenize(text)$ function is used to get the list of the words of each text.

```
1: function SUMMARIZE(text, S)
2:     tokens ← tokenize(text)
3:     for each token ∈ tokens do
4:         S[token] += 1
5:     return S
```

In $divide$, we first combine the two hash maps that consist of the number of occurrences of each word from both sides of the join to get the overall word counts. Next, the $sortByCount()$ function sorts the words by their counts in ascending order and returns a new hash map that has the rank of each word as a value. Finally, the word rank map is put into the PPlan along with the similarity threshold.

```
1: function DIVIDE(S1, S2, SimThreshold)
2:     for each token ∈ S2 do
3:         S1.merge(token, S2.get(token))
4:     TokenRanks ← sortByCount(S1)
5:     PPlan ← (TokenRanks, SimThreshold)
6:     return PPlan
```

In $assign$, we first create a sorted ranked list of words for each text. Then, we calculate the prefix length p [48] 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.

This method aims to assign each text to the fewest possible buckets and choose the rarest words of each text to increase the pruning.

```
1: function ASSIGN(text, PPlan)
2:     tokens ← tokenize(text)
3:     tokenRanks ← ∅
4:     for each token ∈ tokens do
5:         tokenRanks.add(PPlan.TokenRanks.get(token))
6:     l ← len(tokens)
7:     prefixLength ← (l − ceil(C.SimThreshold * l)) + 1
8:     bucketIds ← copyRange(sort(tokenRanks), prefixLength)
9:     return bucketIds
```

Finally, in verify we calculate the Jaccard Similarity of the two sides and return true if they are above the desired similarity threshold.

```
1: function VERIFY(bId1, text1, bId2, text2, PPlan)
2:     tokens1 ← tokenize(text1)
3:     tokens2 ← tokenize(text2)
4:     similarity ← (|tokens1 ∩ tokens2| ÷ |tokens1 ∪ tokens2|)
5:     return similarity > PPlan.SimThreshold
```

C. Overlapping Intervals FUDJ

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

```
1: function SUMMARIZE(interval, S)
2:     S.minStart ← min(S.minStart, interval.start)
3:     S.maxEnd ← max(S.maxEnd, interval.end)
4:     return S
```

In the $divide$ function, we first combine two sides' summaries and unify both timelines. Next, we divide the timeline into $NumberOfBuckets$ and calculate the length of each bucket. Finally, we put all the information required to assign records to the partitions together into $PPlan$.

```
1: function DIVIDE(S1, S2, NumberOfBuckets)
2:     Range.minStart ← min(S1.start, S2.start)
3:     Range.maxEnd ← max(S1.end, S2.end)
4:     length ← (Range.maxEnd − Range.minStart)
5:     d ← length/NumberOfBuckets
6:     PPlan ← (Range, d, NumberOfBuckets)
7:     return PPlan
```

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

```
1: function ASSIGN(interval, PPlan)
2:     R ← PPlan.Range
3:     start ← (interval.start − R.minStart)/PPlan.d
4:     end ← (ceil(interval.end − R.minStart)/PPlan.d) − 1
5:     bucketId ← (front ≪ 16)|end
6:     return bucketId
```

Bucket matching here is not simply equality. So, here we implement a match function that first extracts the starting and ending granule IDs of each bucket and returns true if the buckets are overlapping.

```
1: function MATCH(bucketId1, bucketId2)
2:     b1Start = bucketId1 ≫ 16
3:     b1End = bucketId1&0xFFFF
4:     b2Start = bucketId2 ≫ 16
5:     b2End = bucketId2&0xFFFF
6:     return (b1Start ≤ b2End) and (b1End ≥ b2Start)
```

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

```
1: function VERIFY(i1, bucketId1, i2, bucketId2, PPlan)
2:     return (i1.start < i2.end) and (i1.end > i2.start)
```

VI. FUDJ Infrastructure

This section presents the components of the FUDJ Framework which relies on the common concepts of built-in functions, UDFs, and query optimization. Section VI-A explains how new join algorithms can be registered through a novel statement "CREATE JOIN". Section VI-B describes how the logic from external join libraries will be linked into the system through proxy built-in functions. Section VI-C discusses how the DBMSs can utilize FUDJs and generate optimized query

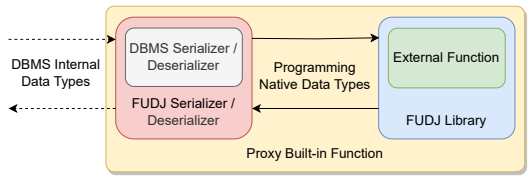


Fig. 7: A Proxy Built-in Function in FUDJ Framework

plans using rewrite rules. Finally, Section VI-D presents the application of FUDJ on Apache AsterixDB.

A. Creating Joins

To facilitate a convenient installation of join libraries, FUDJ introduces a new SQL statement called "CREATE JOIN." For the Apache AsterixDB example, a join library is a JAR package that consists of the classes that implement the FUDJ interfaces. Libraries are uploaded to the system through the terminal. Query 4 is an example of creating a join with the unique name "text_similarity_join" that takes two string keys, and a double similarity threshold as inputs. The external logic for this join is sourced from the "flexiblejoins" library, with the full class name of "textsimilarity.TextSimilarityJoin." 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_similarity_join(a: string , b: string , t:double);
```

Query 4: Create Text-similarity Join

After executing Query 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.

B. Internal and External Actors

An efficient implementation of a join in DBMSs requires access to the internal functionality of the DBMSs. Since this is not a straightforward process for the users, FUDJ aims to translate the external simple implementations into efficient internal functions by extending the concept of UDFs. 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, since UDFs are primarily supported as scalar functions, queries using UDFs may not achieve the same level of performance as those employing optimized join algorithms due to being processed by NLJ operators.

The FUDJ Framework revamps this principle to facilitate user-defined joins. For each function 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 modifies the evaluator using the information embedded in the signature. Subsequently, it creates the internal actor evaluator and passes the external FUDJ library information. Then in each internal actor, the 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 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 VII-B.

As discussed in Section IV, we have two states to consider: $SUMMARY$ and $PPlan$. 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 $PPlan$, 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.

C. 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. Based on the commonalities of partition-based distributed joins that are discussed in Section III, FUDJ modifies the query plan and adds all required elements for each phase as depicted in Figure 8. Please note that although Figure 8 shows the plan starts with a data scan, the source of the data can be other

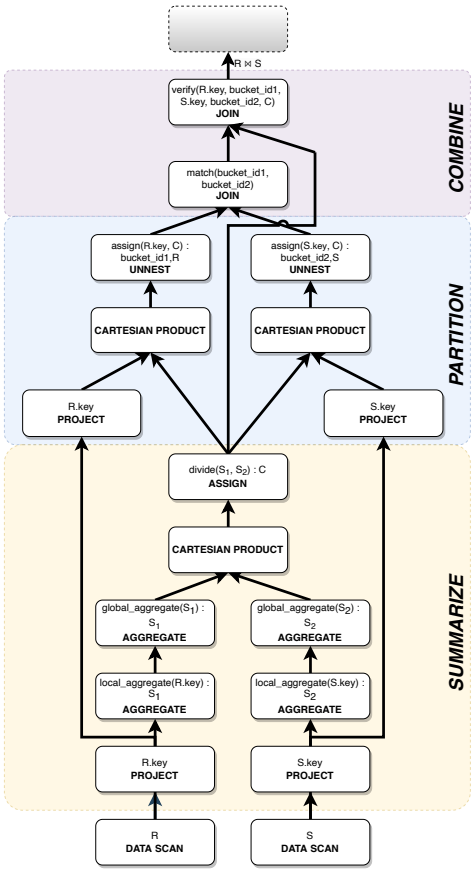


Fig. 8: Flexible User-Defined Distributed Join Logical Plan

operators too. The first group of elements added to the query plan is about the SUMMARIZE phase and they locally and globally aggregate the data and the summaries. The data is first summarized through aggregate operators and The FUDJ query plan starts with the scan of the data and continues with aggregation 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 should also apply physical optimizations when applicable. In this initial design, two further improvements are provided in that context. The first one pertains to self-joins. To let DBMSs optimize self-joins by replicating intermediate results that are used multiple times during query processing, the same proxy built-in functions can be used to handle both sides of the join. 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. 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 FUDJ is overriding the default $summary$ and $assign$ functions. If 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.

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

D. FUDJ Prototype Implementation

To test the feasibility and scalability of FUDJ design, we implemented two prototypes of FUDJ: a single-machine standalone program, which we use for testing and debugging new join implementations, and the other is built on Apache AsterixDB, which tests the scalability on a distributed DBMS engine. Notice that a user-provided implementation seamlessly works on both due to our translation layer explained in Section VI-B.

1) AsterixDB Implementation: Apache AsterixDB [49] 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. This section briefly describes how we implemented the FUDJ Framework on Apache AsterixDB by adhering to the implementation guidelines.

Apache AsterixDB provides a variety of built-in functions and supports UDFs for custom implementations. The FUDJ framework enhances it by allowing developers to use Java primitive types. While built-in aggregate functions exist, UDAs are not currently supported. We modified the runtime mechanism to handle external aggregate functions by connecting them through libraries.

Query optimization is done in Apache AsterixDB by incorporating 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 builds the query plan by following the steps described in Section VI-C.

2) Standalone (Single-Machine) Version: One of the biggest challenges for joining algorithm integration into DBMSs is debugging and testing due to the complexity 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 FUDJ, finding the logical bugs or trying new ideas is straightforward. We share Java implementation[1] with this study, but it can also be transformed into another programming language easily.

VII. Experiments

This section evaluates the FUDJ framework applied to Apache AsterixDB, and Spatial, Interval, and Text-similarity FUDJ implementations. The evaluation begins with a productivity assessment of the implementation methods (using the FUDJ framework and as built-in operators). Next, the section demonstrates that FUDJ framework usage causes minimal query processing overhead when compared to the built-in approach. The section continues with the performance and scalability evaluations of three example join implementations and compares them to the on-top solution (NLJ operator with a UDF). Finally, it studies alternative duplicate handling strategies and outlines future directions for the FUDJ framework and programming model by comparing them 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.

TABLE I: Datasets for FUDJ Experiments

Name	Size	#Records	Key Type
Wildfires [50]	22.1 GB	18M	Point
Parks [51]	7.7 GB	10M	Polygon
NYCTaxi [52]	38.8 GB	173M	Interval
AmazonReview [53]	58.3 GB	83M	Text

Datasets: We use four real-world datasets. For spatial join queries, Parks [51] and Wildfires [50] datasets are used, NYCTaxi [52] is used for interval join queries, and AmazonReview [53] is used for text-similarity queries.

Implementations: FUDJ framework and all of the join algorithms are implemented on Apache AsterixDB 0.9.8. The three example join algorithms Spatial, Interval, and Text-similarity that are based on studies [1], [18], [48] are implemented from scratch, and we will refer to them as **built-in** implementations. The **FUDJ versions** of the example join algorithms in Java[2] are shared with this work. Finally, we will use the term **on-top** to refer to join query processing using the NLJ operator. Lastly, the generated query plans for both the on-top and FUDJ versions are inspected. It is confirmed that they benefit from Apache AsterixDB's optimizations, such as predicate pushdown.

```
/* 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 5: Queries for the experiments

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

A. 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 versions, we implement a rewrite rule for the optimizer, an aggregate function to summarize, an unnesting function to assign records to buckets, a built-in function for bucket matching, and a built-in verify function to filter keys pairs and deduplication if necessary. On the other hand, as we explained in Section IV, FUDJ framework empowers the developer to define the logic for each function, allowing for flexibility and customization while significantly reducing the LOC required. Table II shows that

TABLE II: Written Lines-of-codes for Example Join Implementations Using FUDJ Framework and as Built-in Operators

	Implementation Types	
Join Types	FUDJ	Built-in
Spatial	141 loc	1936 loc
Interval	95 loc	1641 loc
Text-similarity	231 loc	1823 loc

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. Please note that with the LOC metric, we are not comparing FUDJ against the use of programming paradigms in distributed systems. This is because the resulting applications cannot be directly integrated into DBMSs. Therefore, it is still necessary to implement the algorithms from scratch.

Furthermore, reduced LOC in FUDJ versions boosts productivity and streamlines debugging, testing, and code reviewing.

[1]https://github.com/akilsevim/FUDJ-Single-Machine
[2]https://github.com/akilsevim/FUDJ

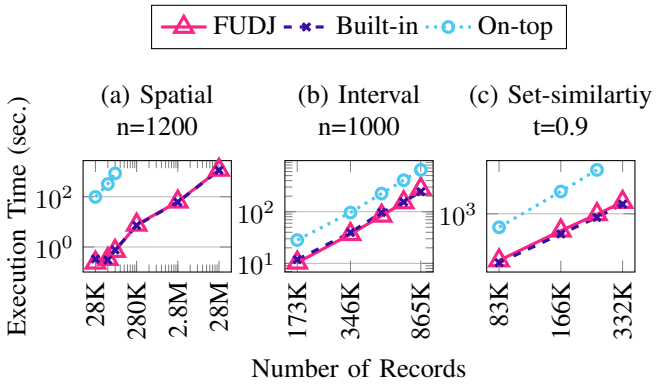


Fig. 9: Join Performance of FUDJ, Built-in, and On-top

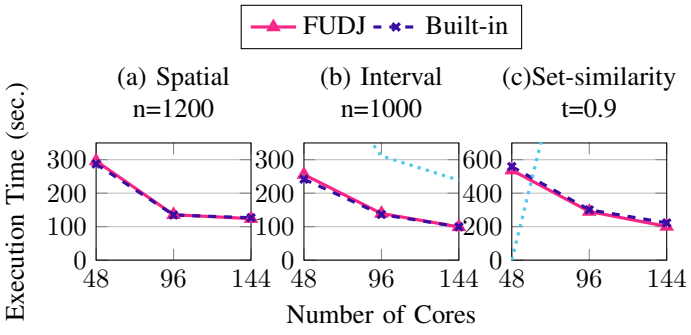


Fig. 10: FUDJ Query Execution Times vs Number of Cores

With fewer lines to manage, developers can pinpoint issues more easily, expediting the debugging phase. Moreover, users' control over FUDJ function logic allows fine-tuning for specific testing scenarios, enhancing application robustness and reliability. These advantages underscore FUDJ's efficacy in distributed programming and database internals.

The integration of new join algorithms into traditional DBMSs often incurs significant deployment costs. Typically, after finalizing the implementation, DBMS software needs to be rebuilt, a process taking approximately 5 minutes in our experimental environment. However, in distributed systems, deploying the rebuilt package to each node adds further complexity and time. Additionally, DBMS often requires stopping and rerunning, causing disruptions. In contrast, FUDJ offers a distinct advantage. It eliminates the need for extensive deployment procedures, allowing swift deployment of new FUDJ packages within seconds without system disruption, making it an efficient choice for introducing new join algorithms.

B. Performance

Figure 9 shows the evaluation of the three implementation methods run on 12-core for a variety of data sizes. Here we run queries using a 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 $\times$ 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 are useful when it comes to the 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-ups had to be measured for small datasets. Hence, the speed up compared to the Spatial FUDJ seems 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.

Figure 9 also shows that the overhead caused by the FUDJ extensible framework is minimal. The difference between FUDJ and Built-in methods for Spatial and Interval joins is 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 objects as Hash Maps.

C. Scalability

To 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 10 shows that Spatial and Text-similarity FUDJ algorithms scale well as compared to the on-top approach. Furthermore, the difference between the built-in and FUDJ implementations remains limited as we increase the number of cores and the data size. As a result, FUDJ does not cause 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 results in the NLJ operator used during the partition matching phase. Since there is no partitioning mechanism for Theta Join in Apache AsterixDB, this operation requires one side to be randomly partitioned resulting in performance degrading. We acknowledge this limitation and are developing an efficient Theta Join operator for future enhancements.

D. 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 the similarity threshold affects the Text-similarity FUDJs performance.

1) Number of buckets: Deciding 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 III, this step is crucial and a

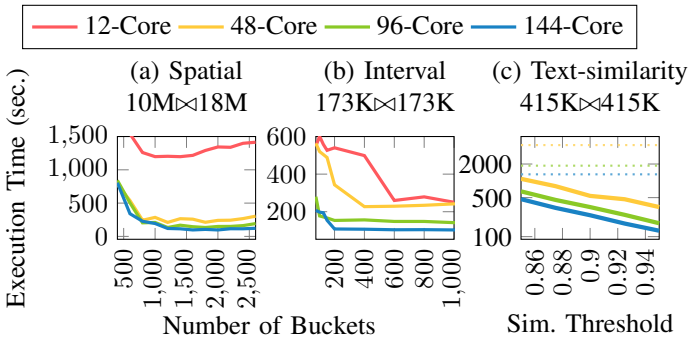


Fig. 11: Effect of Num. of Buckets and Similarity Threshold

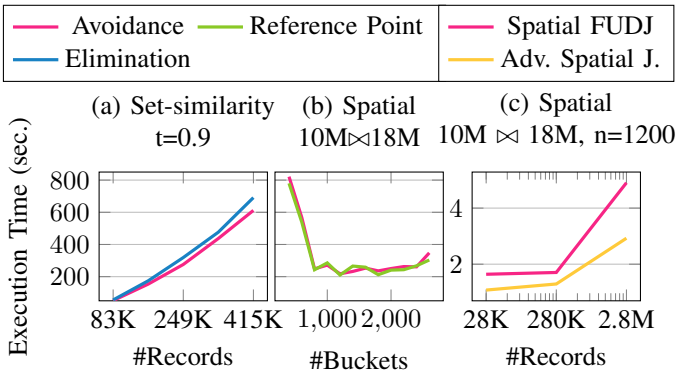


Fig. 12: Duplicate Handling Strategies, and FUDJ and Advanced Optimized Spatial Join Comparison

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. The query execution times are shown in Figure 12.

2) Similarity threshold: On the other hand, although Text similarity FUDJ does not require that the number of buckets is determined, the characteristics of the dataset and most importantly the *similarity threshold* are the main factors for the execution performance. Furthermore, due to the duplication and prefix filtering method, it starts to lose its benefits for low thresholds as can be seen from Figure 12. We used the best-performing number of buckets for the rest of the 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.

E. Duplicate Handling Methods

Duplicate handling is an important aspect of multi-assign optimized join algorithms as discussed in Section III. 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 [48]. In this section, we first test the performance of these two methods on Text-similarity join. Figure 12 shows that Duplicate Avoidance outperforms Duplicate Elimination in all of the dataset sizes by providing 1.15x speedup on average. The FUDJ programming model also allows the developers to implement their own Duplicate Avoidance methods. For instance, in Spatial FUDJ, we implement the Reference Point method described in [18] 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 12b shows that there is not any notable difference between the Reference Point 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.

F. The Effect of Local Join Optimizations

Finally, we will discuss the performance improvement potential of FUDJ by comparing it to existing work such as [4], [18], which involve advanced optimization techniques like plane-sweep. For this purpose, we implemented a highly customized Spatial Join Operator on Apache AsterixDB. The main advantage of this operator compared to the FUDJ version is its ability to apply local optimizations while joining the buckets. Specifically, it employs the plane-sweep method by first sorting the geometries in each tile and then applying spatial merging to efficiently join geometries within each tile. Figure 12c illustrates that having local optimization for spatial joins yields a 1.38x speedup on average. We will further explore this area and propose new operators enabling developers to implement custom local joining mechanisms for additional optimizations.

VIII. Conclusions and Future Work

By offering FUDJ, a system can greatly simplify the way that distributed join algorithms are implemented in data analysis. Such a system would empower users with varying levels of expertise to efficiently leverage efficient purpose-designed join algorithms, significantly reducing the code and knowledge required for their 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, the system would facilitate more comprehensive data analysis, help users uncover hidden insights, and drive accurate decision-making in diverse applications.

In the future, we plan to further enhance our system by adding support for sort-merge-based distributed joins and local join optimizations, such as plane-sweep. Additionally, we aim to automate the process of finding the optimum number of buckets by gathering more dataset statistics during the SUMMARIZE phase. Furthermore, we intend to introduce a Ternary Join Operator to combine MATCH and VERIFY operations, as well as a Theta Join Operator to enhance SAMJ processing for non-equality-based bucket matching.

REFERENCES

[1] A. Dignös, M. H. Böhlen, and J. Gamper, "Overlap interval partition join," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 1459–1470. [Online]. Available: https://doi.org/10.1145/2588555.2612175

[2] N. Ta, G. Li, Y. Xie, C. Li, S. Hao, and J. Feng, "Signature-based trajectory similarity join," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 4, pp. 870–883, 2017. [Online]. Available: https://doi.org/10.1109/TKDE.2017.2651821

[3] L. Chen, S. Shang, C. S. Jensen, B. Yao, and P. Kalnis, "Parallel semantic trajectory similarity join," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 997–1008. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00091

[4] P. Bouros and N. Mamoulis, "A forward scan based plane sweep algorithm for parallel interval joins," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1346–1357, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1346-bouros.pdf

[5] T. Hütter, N. Augsten, C. M. Kirsch, M. J. Carey, and C. Li, "JEDI: these aren't the JSON documents you're looking for?" in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1584–1597. [Online]. Available: https://doi.org/10.1145/3514221.3517850

[6] N. Karpov and Q. Zhang, "Syncsignature: A simple, efficient, parallelizable framework for tree similarity joins," *Proc. VLDB Endow.*, vol. 16, no. 2, p. 330–342, oct 2022. [Online]. Available: https://doi.org/10.14778/3565816.3565833

[7] H. Yuan and G. 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*. IEEE, 2019, pp. 1262–1273. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00115

[8] Z. Shang, G. Li, and Z. Bao, "DITA: distributed in-memory trajectory analytics," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 725–740. [Online]. Available: https://doi.org/10.1145/3183713.3183743

[9] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 495–506. [Online]. Available: https://doi.org/10.1145/1807167.1807222

[10] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman, "Fuzzy joins using mapreduce," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 498–509. [Online]. Available: https://doi.org/10.1109/ICDE.2012.66

[11] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng, "Massjoin: A mapreduce-based method for scalable string similarity joins," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE Computer Society, 2014, pp. 340–351. [Online]. Available: https://doi.org/10.1109/ICDE.2014.6816663

[12] D. Deng, G. Li, H. Wen, and J. Feng, "An efficient partition based method for exact set similarity joins," *Proc. VLDB Endow.*, vol. 9, no. 4, pp. 360–371, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p360-deng.pdf

[13] J. Karimov, T. Rabl, and V. Markl, "Ajoin: Ad-hoc stream joins at scale," *Proc. VLDB Endow.*, vol. 13, no. 4, p. 435–448, dec 2019. [Online]. Available: https://doi.org/10.14778/3372716.3372718

[14] S. A. Shaikh, K. Mariam, H. Kitagawa, and K.-S. 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*, ser. CIKM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3149–3156. [Online]. Available: https://doi.org/10.1145/3340531.3412761

[15] Y. N. Silva, W. G. Aref, and M. H. Ali, "The similarity join database operator," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, 2010, pp. 892–903.

[16] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. Borkar, M. J. Carey, and C. Li, "Similarity query support in big data management systems," *Information Systems*, vol. 88, p. 101455, 2020.

[17] J. Carman, Eldon P., "Interval joins for big data," Ph.D. dissertation, 2020. [Online]. Available: https://www.proquest.com/dissertations-theses/interval-joins-big-data/docview/2458188626/se-2

[18] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 1996, pp. 259–270. [Online]. Available: https://doi.org/10.1145/233269.233338

[19] M. Stonebraker, J. Anton, and M. Hirohama, "Extendability in POSTGRES," *IEEE Data Eng. Bull.*, vol. 10, no. 2, pp. 16–23, 1987. [Online]. Available: http://sites.computer.org/debull/87JUN-CD.pdf

[20] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized search trees for database systems," in *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, 1995, pp. 562–573. [Online]. Available: http://www.vldb.org/conf/1995/P562.PDF

[21] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. 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*. IEEE Computer Society, 2009, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CLUSTR.2009.5289178

[22] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania, "Processing multi-way spatial joins on map-reduce," in *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, G. Guerrini and N. W. Paton, Eds. ACM, 2013, pp. 113–124. [Online]. Available: https://doi.org/10.1145/2452376.2452390

[23] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015, pp. 34–41.

[24] E. H. Jacox and H. Samet, "Iterative spatial join," *ACM Trans. Database Syst.*, vol. 28, no. 3, p. 230–256, sep 2003. [Online]. Available: https://doi.org/10.1145/937598.937600

[25] P. Bouros and N. Mamoulis, "Spatial joins: What's next?" *SIGSPATIAL Special*, vol. 11, no. 1, p. 13–21, aug 2019. [Online]. Available: https://doi.org/10.1145/3355491.3355494

[26] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, p. 7–es, mar 2007. [Online]. Available: https://doi.org/10.1145/1206049.1206056

[27] X. Zhou, D. J. Abel, and D. Truffet, "Data partitioning for parallel spatial join processing," *Geoinformatica*, vol. 2, pp. 175–204, 1998.

[28] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *2015 31st IEEE International Conference on Data Engineering Workshops*, 2015, pp. 34–41.

[29] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 131–140. [Online]. Available: https://doi.org/10.1145/1242572.1242591

[30] L. A. Ribeiro and T. Härder, "Generalizing prefix filtering to improve set similarity joins," *Information Systems*, vol. 36, no. 1, pp. 62–78, 2011, selected Papers from the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437910000657

[31] W. Mann and N. Augsten, "PEL: position-enhanced length filter for set similarity joins," in *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014*, ser. CEUR Workshop Proceedings, F. Klan, G. Specht, and H. Gamper, Eds., vol. 1313. CEUR-WS.org, 2014, pp. 89–94. [Online]. Available: https://ceur-ws.org/Vol-1313/paper_16.pdf

[32] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and

A. Fuxman, Eds. ACM, 2012, pp. 85–96. [Online]. Available: https://doi.org/10.1145/2213836.2213847

[33] Y. N. Silva, S. S. Pearson, J. Chon, and R. Roberts, "Similarity joins: Their implementation and interactions with other database operators," *Information Systems*, vol. 52, pp. 149–162, 2015, special Issue on Selected Papers from SISAP 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437915000186

[34] P. Bakalov, M. Hadjieleftheriou, E. Keogh, and V. J. Tsotras, "Efficient trajectory joins using symbolic representations," in *Proceedings of the 6th International Conference on Mobile Data Management*, 2005, pp. 86–93.

[35] P. Bakalov and V. J. Tsotras, "Continuous spatiotemporal trajectory joins," in *GeoSensor Networks: Second International Conference, GSN 2006, Lecture Notes in Computer Science, vol 4540*. Springer, 2008, pp. 109–128.

[36] Y. Chen and J. M. Patel, "Design and evaluation of trajectory join algorithms," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2009, pp. 266–275.

[37] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis, "Parallel trajectory similarity joins in spatial networks," *The VLDB Journal*, vol. 27, no. 3, pp. 395–420, 2018.

[38] S. Wang, Z. Bao, J. S. Culpepper, and G. Cong, "A survey on trajectory data management, analytics, and learning," *ACM Comput. Surv.*, vol. 54, no. 2, mar 2021. [Online]. Available: https://doi.org/10.1145/3440207

[39] A. Dignös, M. H. Böhlen, J. Gamper, C. S. Jensen, and P. Moser, "Leveraging range joins for the computation of overlap joins," *VLDB J.*, vol. 31, no. 1, pp. 75–99, 2022. [Online]. Available: https://doi.org/10.1007/s00778-021-00692-3

[40] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, p. 1016–1027, jun 2012. [Online]. Available: https://doi.org/10.14778/2336664.2336674

[41] A. Shahvarani and H. Jacobsen, "Distributed stream KNN join," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 1597–1609. [Online]. Available: https://doi.org/10.1145/3448016.3457269

[42] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, S. D. Gribble and D. Katabi, Eds. USENIX Association, 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[44] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 119–130.

[45] M. Carey and L. Haas, "Extensible database management systems," *ACM SIGMOD Record*, vol. 19, no. 4, pp. 54–60, 1990.

[46] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-26253-2

[47] M. Bandle, J. Giceva, and T. Neumann, "To partition, or not to partition, that is the join question in a real system," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 168–180. [Online]. Available: https://doi.org/10.1145/3448016.3452831

[48] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. R. Borkar, M. J. Carey, and C. Li, "Similarity query support in big data management systems," *Inf. Syst.*, vol. 88, 2020. [Online]. Available: https://doi.org/10.1016/j.is.2019.101455

[49] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann, "Asterixdb: A scalable, open source BDMS," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1905–1916, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf

[50] 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.

[51] A. Eldawy and M. 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.

[52] C. 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.

[53] R. He and J. J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, J. Bourdeau, J. Hendler, R. Nkambou, I. Horrocks, and B. Y. Zhao, Eds. ACM, 2016, pp. 507–517. [Online]. Available: https://doi.org/10.1145/2872427.2883037