Siren Federate: Bridging the Gap between Document and Relational Data Systems for Efficient Exploratory Graph Analysis

Stéphane Campinas, Matteo Catena, and Renaud Delbru

Siren – Galway, Ireland
name.surname@siren.io
https://siren.io/

Abstract. Investigative workflows requires interactive exploratory analysis on large heterogeneous knowledge graphs. Current databases show limitations in enabling such task. This paper discusses the architecture of Siren Federate, a system that efficiently supports exploratory graph analysis by bridging the document-oriented and relational models. Technical contributions include distributed join algorithms, adaptive query planning, query plan folding, and semantic caching. Experiments show that Siren Federate exhibits low latency and scales well with the amount of data, the number of users, and the number of computing nodes.

Keywords: Exploratory Graph Analysis · Knowledge Graph · Database and Information System Architecture · Distributed Join Algorithms · Document-oriented Database.

1 Introduction

Siren provides its Investigative Intelligence platform to Law Enforcement, National Security and Cyber-threat investigators. Investigative intelligence is a specialized area of data analytics with the goal of uncovering threats and criminal activities [1,3] through the analysis of inter-connected data.

Knowledge graphs [13] are fundamental elements in investigation systems, as they integrate diverse data into a unified graph for data analysis. In fact, investigations often involve connecting the dots across large amount of structured (such as database table records), semi-structured (such as XML, JSON, or logs), and unstructured data (such as text or multimedia content). Malicious actors exploit the increasing volume and complexity of these data to blend in and operate. Therefore, it is essential that exploratory graph analysis works at scale to uncover complex chains of dependencies hidden within massive volumes, thus allowing to trace incidents, detect vulnerabilities, and manage risks. Additionally, investigative systems must have fast response times: investigators often interact with the system through an explorative and iterative process, which can be hindered by large system latency [18].

Current database systems face scalability and flexibility challenges implementing these requirements, as detailed in Section 2. Siren Federate addresses these challenges by integrating relational and graph analytics into Elasticsearch, a distributed Information Retrieval (IR) system [10]. Its extension enables efficient analysis of massive knowledge graphs while retaining searching and relevance ranking capabilities of IR systems.

In this paper, we discuss how Siren Federate bridges the gap between document-oriented and relational models, we illustrate its architecture, and finally we evaluate its performance on a large synthetic dataset.

2 Motivations

By mapping data sources to entities (vertices) and their relationships (edges) in a graph structure, knowledge graphs provide a flexible and dynamic framework for data integration and retrieval, crucial in rapidly evolving domains. Investigative workflows on knowledge graphs often involves exploratory analysis [17] for discovering patterns and generating new leads. An investigator may start with limited information and iteratively expand their search within the graph to uncover new evidences. The system must guide users in searching, filtering, and drilling down through this data to pinpoint potential entities of interest. Additionally, graph analytical capabilities such as path finding, centrality, or community detection, can help discovering relevant subgraphs to investigators.

Siren's platform supports this workflow by combining several data interaction paradigms – search, analytic dashboards, set-to-set navigation[1], and graph visualization – into a coherent model. For instance, individuals, cellphone data, calls, texts, and network cells are linked together in a Signals Intelligence scenario to form a complex graph. Set-to-set navigation guides investigators in connecting together those different datasets. Applying filters to one set impacts all (in)directly connected sets, allowing exploration of the relevant information. The investigator can move from cellphones to related records, like locations visited by cellphone's owners whom received crime-related texts. Graph visualization helps visualizing the inter-connected sets, identifying patterns or clusters, and answering questions like *"Which people own which cellphone? Do they connect to the same network cell? Do they meet with other groups of users at other times?"*.

Effective exploratory graph analysis must handle diverse query workloads: (a) searching textual documents (e.g., social media, open web, mobile forensic data) and arbitrary records with relevance ranking and highlighting; and (b) relational and graph database workloads [4] such as OLTP queries (processing only a localized part of the graph), OLAP queries (spanning large portions of the graph), neighborhood queries, traversal queries over long graph paths, and global graph analytics (e.g., global pattern matching, graph search, community detection, path finding, centrality). The challenge is to maintain such a mixed workload at large scale and interactive speed, with response times ranging from sub-seconds to seconds to not impact the investigator's workflow.

Despite the capabilities of graph and relational databases, they often fall short in meeting investigative workflows requirements [4]. Native graph stores

[1] Set-to-set navigation is a type of relational faceted navigation [23].

struggle with large-scale data due to limited sharding and replication. Relational databases handle well structured data and complex queries, but lack flexibility for heterogeneous data and have limited graph, text processing, and search capabilities [7]. Multi-model databases attempt to unify various data models, but their origins in specific models often lead to inefficiencies in handling diverse workflows. Achieving optimal performance across relational, graph, and full-text search remains difficult due to challenges in query processing, schema design, and indexing [20]. Finally, polyglot architectures do not fully resolve these issues, as they require data duplication and movement across multiple specialized backends [21]. This led us to use Elasticsearch as it supports a flexible data model, scales well on commodity machines, and offers advanced text processing and search capabilities. Elasticsearch offers interactive speed by leveraging inverted indices for search, columnar storage for fast analytics, and effective caching strategies.

However, document-oriented databases like Elasticsearch have limitations in joining data. Joins must be pre-planned at indexing time, storing documents to be joined on the same index shard.[2] Without resorting to data duplication, such a mechanism is only suitable for hierarchical relationships but not for more complex ones like networks. In fact, joins are needed to implement a competent graph analytics system [32]. Therefore, a distributed architecture that combines the strengths of graph databases, relational databases, and IR systems is needed. It must scale to massive heterogeneous graphs and support efficient relational and graph operations, to enable the iterative exploratory analysis of intelligence workflows. Siren Federate addresses these challenges by incorporating query-time distributed join capabilities between different indices into Elasticsearch.

3 Bridging the Document-Oriented and Relational Models

In a document-oriented store, one approach to model a graph is to map vertices and edges to documents [4]. IR systems provide flexible data modeling and advanced search capabilities, but lack the necessary relational join operations required by exploratory graph analysis (e.g., to find adjacent vertices). Siren Federate bridges this gap by allowing joins within the document-oriented model, thus supporting the analysis of knowledge graphs.

Several works attempt to bridge between document-oriented and relational models by mapping the first into the second [6,14,27,31]. However, these approaches are limited to what relational databases propose and miss optimizations offered by IR systems for processing and searching documents. Siren Federate takes the opposite approach, mapping from a relational data model to a document-centric data model in order to fully leverage what IR systems offer.

In the relational model, a join combines rows from multiple tables into a new table. Instead, in the document-oriented model, queries are applied to documents in an index, and returns matching documents. Siren Federate expresses a join operation $\bowtie$ within this model as the process of finding documents from

[2] https://www.elastic.co/guide/en/elasticsearch/reference/8.13/joining-queries.html

an index (the parent set) that are related to documents from another index (the child set) according to specific conditions, like field equality. Siren Federate implements (a) the *semi-join* $\ltimes$ for filtering the parent set's documents based on the child set's documents; and (b) the *inner-join* $\bowtie$ for extending the parent set's documents with fields from the matching child set's documents. These mechanisms are well suited for the iterative exploration needed in investigative intelligence, allowing to refine a target set of documents with results from previous investigation stages.

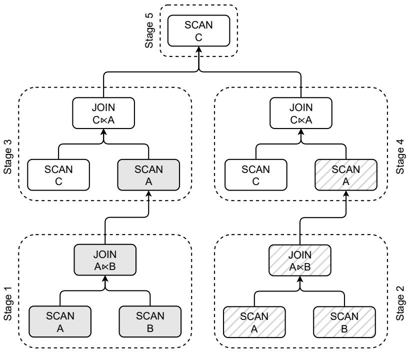

Fig. 1: A staged logical query plan

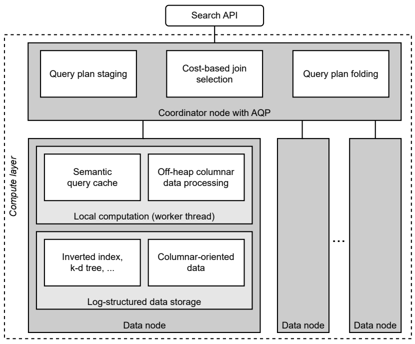
Search API
Query plan staging
Cost-based join selection
Query plan folding
Coordinator node with AQP
Compute layer
Semantic query cache
Off-heap columnar data processing
Local computation (worker thread)
Inverted index, k-d tree, ...
Columnar-oriented data
Log-structured data storage
Data node
...
Data node
Data node

Fig. 2: Siren Federate's architecture

The logical steps for Siren Federate to join two document sets from indices A and B are highlighted in gray in Fig. 1. Two of the steps involve a SCAN operation, searching over the parent set A and child set B to retrieve subsets of documents to be joined. These documents may need to be exchanged across the computing cluster according to one of the different strategies described in Sec. 4.3. The parent and child subsets are then locally joined on the cluster's nodes by a JOIN operation, using data structures like hash tables, inverted indices, or k-d trees. The join results are tuples (in the relational sense) representing documents from the parent index that have fulfilled the join conditions. These tuples are then used by another SCAN operation to filter the parent index, retrieving the parent documents that meet the join conditions. This model also supports multi-join operations, with multiple child sets joined with the parent set. This is represented using a non-binary tree structure, where each SCAN operation can be associated with more than one child JOIN operation (see Fig. 1).

Siren Federate follows a *late materialization* approach, scanning only fields from parent documents to evaluate a join operation and avoid manipulating entire documents. Each document is associated with a global ID (see Sec. 4.1), to uniquely identify it across the system. Tuples produced by the join operation include this ID, rather than the entire document content. Upstream operations can use this ID to materialize necessary fields. This strategy is used for various operations, such as filtering, sorting, aggregating, and retrieving document content. These operations are delegated to the underlying Elasticsearch engine, which is optimized for handling such tasks efficiently.

Given the document-centric model, tuples produced by the join must be grouped and sorted by the global document ID, as the join may produce scattered tuples about the same parent document, for example in the case of many-to-many relationships. This enables the parent SCAN operation to efficiently merge the join output based on ordered document IDs, which align with the natural order of the underlying log-structured storage (see Sec. 4.1). We employ efficient exchange strategies for optimizing the grouping and sorting operations (see Sec. 4.3).

Siren Federate uses this logical model to integrate relational joins into the document-oriented model. This representation drives the architecture design and runtime behavior of Siren Federate. For instance, the adaptive query planner uses it to stage the query plan execution. The semantic information embedded within this model is used by the semantic caching, but also for folding the query plan. Finally, this model retains the search engine's capabilities to efficiently execute filters, sorting, and aggregations.

4 Siren Federate Architecture

This section introduces the core architectural components of Siren Federate, shown in Fig. 2. Siren Federate serves as the compute layer of an investigative system, leveraging the distributed computing and storage architecture of Elasticsearch for scalability. The application layer of the investigative system relies on Siren Federate's relational and analytical capabilities via its search API.

The distributed IR system consists of a cluster of computing nodes. Each node plays a different role in the cluster: coordinator nodes are responsible for planning the execution of a request received from the search API, while data nodes are responsible for storing data and executing operations dictated by the coordinator's query plan. This architecture ensures sub-second to seconds response time at scale, as computational load is distributed across data nodes, which can independently process the log-structured data storage to produce results (Sec. 4.1). Data nodes execute scan and join operations using a columnar data processing model (Sec. 4.2) and different join algorithms (Sec. 4.3). The query plan defined by the coordinator is divided into multiple stages (Sec. 4.4). Redundant operations of the query plan are folded to avoid unnecessary computation (Sec. 4.5). The logical query plan is processed iteratively, stage-by-stage, interleaving its physical planning with its execution. At each iteration, a cost-based query optimizer checks the semantic cache to reuse existing join results (Sec. 4.6) or selects the most efficient join algorithm.

4.1 Log-Structured Distributed Data Store

Siren Federate leverages Elasticsearch's distributed data store, which horizontally partitions data across nodes using document sharding. An index is partitioned into shards, and each document is routed to a shard. A shard is a Lucene index [11], based on a log-structured model [22], and composed of one or more index segments. The log-structured model adopts an append-only update strategy and consists in creating a file-based data structure called index segment. Segments are immutable and get merged over time or when a size threshold is reached. This append-only model allows for (1) implementing a lightweight read-lock mechanism to guarantee data consistency during the execution of distributed joins, enabling the concurrent execution of queries and real-time data updates; and (2) dynamically generating a global ID for documents by combining shard and segment IDs with the document's insertion order, thanks to the immutability of segments. This global ID enables the quick location of a document's physical position in the cluster, and is leveraged to achieve late materialization during the computation of join operations, as explained in Sec. 3.

4.2 Columnar In-Memory Processing

Siren Federate stores data for intermediate join computation into off-heap main memory using a columnar layout and leverages compression algorithms optimized for specific data types. During join operations, Siren Federate processes only the relevant fields, such as join key fields and global document IDs. The data exhibits a tabular structure, with tuples corresponding to documents and columns to their fields. There are two approaches for processing tabular data: row-at-a-time and column-at-a-time.

The row-at-a-time approach reads whole tuples even if only a few columns are needed, leading to CPU cache misses and negatively impacting the performance. Following best practices from [15], Siren Federate adopted the column-at-a-time approach, improving the query performance by a factor of 2 compared to the row-at-a-time implementation.[3]

The column-at-a-time approach uses a batch-processing pipeline. Each batch stores a fixed number of tuples, stored in a columnar fashion. The size of a batch is optimized to fit within the CPU cache line to avoid cache misses. Although batches are currently processed sequentially by a worker thread, future plans include parallel batch processing to increase throughput. A profiling tool[4] showed an increase of the CPU cache usage with the column-at-a-time approach: Siren Federate ver. 27.5 increased "cache-references" by 25% compared to ver. 22.6 that uses the row-at-a-time approach, demonstrating enhanced CPU cache utilization.

4.3 Distributed Join Algorithms

Siren Federate implements join techniques that leverage the intrinsic data structures of the underlying IR system to ensure scalability and high performance. An example with two distributed indices A and B is shown in Fig. 3. Both indices are partitioned into three shards, whose data needs to be exchanged across the computing nodes in order to be joined. The available join strategies are

[3] https://info.siren.io/content/siren-benchmark-whitepaper

[4] https://github.com/async-profiler/async-profiler

Broadcast Hash Join Data from the child index is forwarded to all computing nodes hosting shards of the parent index (see Fig. 3, left). Local hash tables, created from the received data, are probed while scanning the parent index's columnar storage. Worker threads process segments in parallel, and local hash tables are shared across these worker threads.

Broadcast Index Join This strategy utilizes Lucene's inverted indexes (akin to burst tries [12]) for binary values, and Bkd-trees [26] for numerical values. Data are exchanged like the broadcast hash join (see again Fig. 3, left), but the child set data is used for index lookups over the parent set, eliminating exhaustive scans of the columnar storage. Worker threads process segments and probe the index with the received data. This is effective for graph expansion or path finding tasks, where the objective is to incrementally expand relationships from a group of records.

Partitioned Hash Join Inspired by [30], it leverages the columnar storage to scan data from the parent and child indices, partitioning data across computing nodes, and creating localized hash tables for each partition (see Fig. 3, middle). This method employs morsel-driven parallelism and involves a two-step partitioning to create fixed-sized work units: an initial node partitioning at the scan level (sender side) and a second partitioning at the join level (receiver side). This method achieves better parallelism and reduced memory and network overheads compared to strategies like the broadcast hash join. In Fig. 3 only three computing nodes are shown for the sake of space. However, this join strategy can leverage all cluster nodes regardless the number of shards.

Routing Join Similar to the broadcast hash join, it leverages the document sharding to reduce network traffic. It reuses the sharding routing function of the parent index to partition and exchange the child set's tuples to the corresponding parent set's shards [5] (see Fig. 3, right). Each worker thread employs either a hash table-based strategy (like the broadcast hash join) or an inverted index-based strategy (like the broadcast index join) to compute the results. Preliminary experiments (not presented in this work) indicate a 30% reduction in response times compared to the broadcast hash join strategy.

These strategies optimize specific scenarios. The role of the query planner (Sec 4.4) is to select the most cost-effective join strategy by considering factors such as shard topology and set cardinality to optimize the cluster's utilization.

4.4 Adaptive Query Planner

Accurate join cardinality estimation is crucial for planning the most effective join algorithm. Unfortunately, this is challenging with complex query plans with deeply nested joins, common in investigative scenarios. Traditional static methods, based on histograms and simple cardinality estimation formulas, often yield inaccurate estimations due to assumptions like attribute independence and distribution uniformity, resulting in sub-optimal selection of join algorithms. These inaccuracies are exacerbated as the complexity of the query plan increases [16]. Index-based join sampling, while more accurate, is computationally expensive,

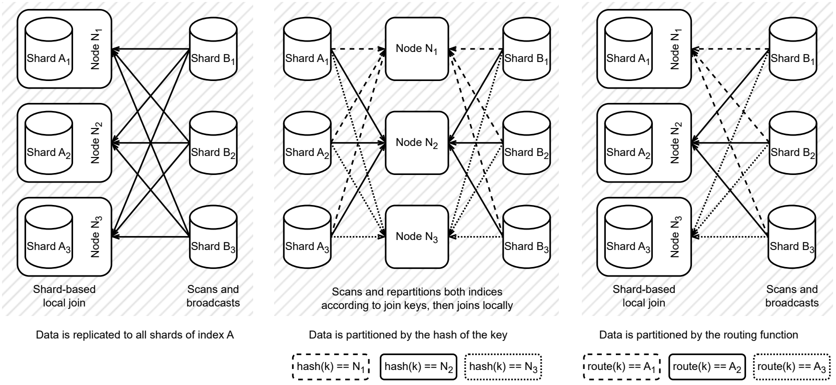

Fig. 3: Federate's distributed join algorithms: (left) Broadcast Hash/Index Join, (middle) Partitioned Hash Join, (right) Routing Join. Arrows represent data exchange between computing nodes

especially in distributed systems where it requires data shuffling across the network, and also suffers from inaccuracies with long sequence of joins.

To address this, Siren Federate implements an adaptive query planner (AQP) that interleaves planning and execution via stages [9]. This approach collects runtime statistics during execution, allowing more accurate cardinality estimation compared to static methods, especially for long sequence of joins. This enables to dynamically adjust the query plan based on real-time feedback. AQP operates in several key phases:

Logical Plan Generation The planner generates a logical query plan divided into stages. Each stage corresponds to a materialization point where an intermediate result is fully created before proceeding further. Typically, it comprises a logical join and two logical scans.
Physical Optimization The planner gathers statistical information, computes costs for various join strategies, and selects the optimal one. This is repeated for each stage, leveraging runtime cardinality estimates from already computed nested joins (stages).
Execution The physical sub-graph of each stage is executed, materializing intermediate results before proceeding.
Parallelization The query plan enables parallel execution of independent stages. Independent stages are executed concurrently, while dependent stages must wait for predecessors to complete.

To illustrate how AQP works, consider a dataset with three indices. An index A of documents representing cellphones, with fields containing the phone number, the operator, reference to the person who subscribed the contract, etc. An index B of documents representing the online activity of a person such as social media posts, with fields like person's identity and textual content. An index C of documents representing each call detail record (CDR) with fields such as time, duration, completion status, source and destination numbers of a call [29]. Imagine we want to find all CDRs related to phones used by people involved in suspicious online activities. AQP would generate a logical query plan in stages (logical plan generation) as shown in Fig. 1. Assume filters (e.g., keyword matching or vector search) applied to B identifies crime-related posts, then the set of "phones used by people involved in suspicious online activities" is the result of $A \ltimes B$ (Stages 1 and 2). The set of CDRs where these phones are the caller is returned by $C \ltimes A$ using the CDR's caller as the join key (Stage 3). Similarly, the set of CDRs where these phones are the callee is returned by $C \ltimes A$ using the CDR's callee as the join key (Stage 4). The disjunction of these two sets produces the desired results (Stage 5). Different join strategies may be used depending on the statistical information gathered from the previous stages (physical optimization and execution). Since Stages 1 and 2 are independent, they can be executed in parallel before moving to Stages 3 and 4 (parallelization).

4.5 Query Plan Folding

User queries often contain redundant operations that negatively impact query processing, such as repeated searches or joins. Redundancies commonly occur when investigating related entities through various graph topologies, boolean expressions, or batched requests targeting the same entities with diverse filters or aggregations. Redundant operations affect also SQL query processing [19,28].

To address this challenge, Siren Federate adopts a query plan folding that uses the semantic definition of query operators to detect and merge redundant operators across one or more logical query plans. The semantic definition of an operator captures its logical meaning, structure, dependencies, and state of the data tables it involves [8].

Siren Federate handles not only the folding of selection and scan operations [28], but also of join operations. The folding strategy consolidates redundant operations into a unified shared operator. In the previous AQP example, the operators from Stage 1 and its subsequent SCAN A are folded with those from Stage 2 and SCAN A (highlighted by a hatched background in Fig. 1) since they represent the same join. However, Stage 3 and 4 are not folded as they have different join conditions: the CDR's caller number is used in Stage 3, while the CDR's callee number is used in Stage 4. Similarly, SCAN C is not folded between the two stages because it scans different fields.

4.6 Semantic Caching

In exploratory graph analysis, iterative analysis often results in recurrent execution of the same join operations. By caching these, the system can optimize subsequent related queries, reducing the latency and computational load.

Semi-joins are well-suited for caching compared to other join types, as their outputs can be represented as sets of document IDs. Exploiting this, Siren Federate employs semantic caching, relying on the semantic definition of query operators (Sec. 4.5). Compared to conventional caching methods which operate at the query syntactic level [24], semantic caching [8] indexes cache entries according to query operator semantics, guarantying data consistency and resilience to changes in the underlying data, even when query operators depend on multiple data sources derived from descendant query operators. Unlike traditional caching strategies that focus on raw results which can lead to large memory overhead, the semantic caching strategy uses compact bitset representations to efficiently encode semi-join outputs. This reduces memory consumption and increases the potential amount of cached operations, enhancing the overall system efficiency.

To illustrate this, take the example from Sec. 4.4. In a subsequent iteration, the investigator wants to identify, from a new index D, people owning phones involved in previously found CDRs. This adds a new join $D \ltimes C$, with C being the results of the previous iteration. The AQP generates a logical plan including the subtree from Fig. 1. With semantic caching, the results of the subtree can be reused, meaning that only the additional join $D \ltimes C$ must be computed.

This strategy benefits graph analytics, particularly path finding algorithms which can be represented as sequences of semi-joins. Semantic caching of semi-joins reduces redundant computations, minimizing the number of operations and associated I/O, and resulting in a more efficient execution of graph queries [25].

4.7 Principles for a competent graph analytics system

In [32], ten Wolde et al. identify eight core features necessary for a competent graph analytics system. Siren Federate leverages and extends the capabilities of Elasticsearch to meet these principles:

Fast Scans on Elements with Schema Siren Federate uses Elasticsearch's dynamic schema capabilities and column-oriented storage for fast attribute scanning. Dynamic schema-awareness offers flexibility in handling knowledge graph variability and optimizes query processing by adapting to the data structure.
Skippable Compressed Columnar Storage Elasticsearch supports fast columnar scans with data skipping via pushed-down predicates, combining columnar storage ordered by document IDs with an inverted index or k-d trees.
Vectorized or Data-Centric Execution Siren Federate adopts column-at-a-time (vectorized) processing for its data pipelines during scan and join operations, and an in-memory vector format with compression and data skipping.
Morsel-Driven Multi-Core Siren Federate uses morsel-driven parallelism during scan and join operations to distribute constant-sized work units (morsels) across worker threads, reducing load imbalance and optimizing CPU cache usage. Dynamic index segment splitting is critical for better parallelism during scans of large segments, and reducing query execution latency.
State-of-the-Art Query Optimization Siren Federate's AQP splits query plans into stages and employs runtime estimation to avoid the overhead of traditional table sampling, ensuring efficient query execution.
Bulk APIs/Algebras Elasticsearch's boolean algebra operates as a bulk API for predicate evaluation by enabling manipulation of document sets efficiently. Siren Federate integrates relational algebra that also functions as a bulk API.

Out-of-Core Buffer Manager While performing in-memory join operations, Siren Federate leverages Elasticsearch's ability to handle out-of-core data sizes efficiently during scan. Frequently accessed columns are cached at the operating system level, ensuring scan operations in RAM.
Explicit Control over Memory Locality Siren Federate uses off-heap memory management to reduce garbage collection overhead when handling gigabytes of data in memory for short durations, and optimizes memory locality through columnar storage, morsel-driven parallelism, and effective data partitioning.

To further enhance performance in a distributed environment, Siren Federate adopts three additional core features:

Data Locality Leveraging data locality minimizes data movement across the network, a common bottleneck as it requires additional intermediate serialization and copy of the data. Siren Federate performs late materialization of documents using global IDs to quickly locate documents in the cluster and co-locates data by reusing existing data routing coming from document sharding.
Data Exchange Effective data exchange operators exploit the physical distribution and structure of the storage to maximize memory locality without data reorganization. Preserving the implicit order, even partially, of materialized tuples during scan and join operations improves the performance of the exchange operator, which must group and sort tuples based on the document ID as explained in Sec. 3. Radix partitioning [33] is a highly efficient method for clustering tuples from a range of documents together, improving document sorting.
Caching Implementing compact caching strategies for intermediate results allows reuse across queries and users, reducing redundant computations and improving overall system efficiency, especially in incremental exploratory scenarios.

5 Evaluation

This section evaluates the scalability and performance of Siren Federate in computing semi-joins[5] across various dimensions, such as the number of cluster nodes, concurrent users and data volume.

Dataset. We use a synthetic dataset[6] with 15.6 billion documents tracking the positional information of cell phones to mimic scenarios where analysts monitor phone calls. This dataset covers 100 days, with 156 million documents per day, 6.5 million unique phone identifiers, and 2,400 positions per phone. One Elasticsearch index per day is created, with 8 primary shards with no replicas. The total size of the data is 2.7 TB.

[5] We focus on semi-joins as they are more suitable than inner-joins for large datasets. For semi-joins, the output size is linear with the cardinality N of the parent set, compared to $O(N \times M)$ for inner-joins (M being the cardinality of the child set). Semi-joins are used extensively for exploratory graph analysis tasks like set-to-set navigation, graph expansion, and pathfinding, due to their efficiency and scalability.

[6] https://gist.github.com/scampi/07e7bd556fe016a5cba6c092c3f418fb

Setup. The benchmark tests use AWS "i3.4xlarge" machines with Broadwell processors (16 vCPUs), a local NVME drive for Elasticsearch data, a gp2 drive for the OS, and a 10 Gbps network link. We use Java Virtual Machine 20.0.1, Elasticsearch 8.7.1, and Siren Federate 31.1. Elasticsearch is configured with 30 GB heap memory, and Siren Federate with 16 GB off-heap memory.

Experiments. We evaluate Siren Federate with varying cluster sizes (12 to 36 nodes), data volumes, and concurrent users (1, 5 and 10). The system is setup to serve the maximum number of concurrent users. We use the following queries with different complexity:

Q1 joins phone numbers in a given area on one day with those in another area on another day (78 million documents per set); **Q2** is similar to Q1 but over a week (546 million documents per set); **Q3** , given a phone number, finds other phones at the same location over 90 days (14 billion documents filtered with 2,160 documents); **Q4** finds phones at the same location on two different days (156 million documents each); **Q5** is similar to Q4 but over a week (more than 1 billion documents per set); **Q6** is similar to Q4 but over two weeks (more than 2 billion documents per set).

Queries use the *partitioned hash join*, except Q3, which uses the *broadcast index join* due to the small cardinality of its child set. We measure the execution time for a randomly-selected query with a fixed number of concurrent users, bypassing query caches. The benchmark runs until at least 100 measurements per query are produced, reporting the 90th percentile processing time (P90).

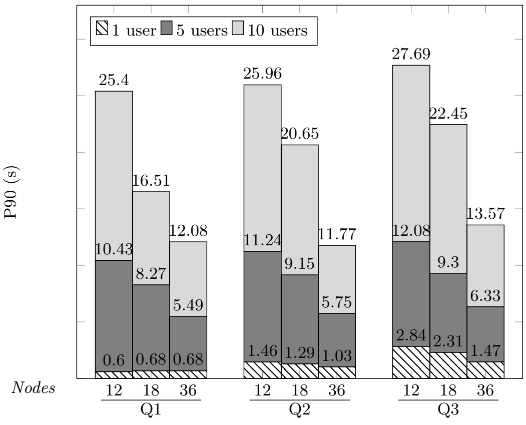

Fig. 4: Query times for queries Q1 to Q3 with varying number of nodes and users

Results. Fig. 4 reports P90 for queries Q1 to Q3. With one concurrent user, Q1 joins a total of 156 million documents in subsecond time. However, P90 does not decrease as the number of nodes increases because the amount of data joined is too small. The latency of the join phase represents an insignificant part of the response time, while the scan phase is tied to a limited number of shards and cannot be further distributed.

Q2 joins over 1 billion documents (×7 more data than Q1), with P90 increasing by at most ×2.5, indicating a better usage of the computing resources. However, we notice that response time does not decrease as expected as the number of nodes increases. In fact, increasing the number of nodes by ×3 only decreases P90 by 30%. We assume this is because the latency of Q1 and Q2 are dominated by some fixed overhead during the query planning and pipeline execution. This requires further analysis.

Q3, which filters 14 billion documents using the *broadcast index join* strategy, shows P90 decreasing by ×2 as nodes increases by ×3. This strategy avoids partitioning and shuffling such a large parent set while still distributing load across the full cluster as the number of nodes increases.

With 5 concurrent users, P90 of Q1 increases by ×12.5 – on average across cluster's configurations – when compared to response times with 1 user; Q2 increases by ×6.8; Q3 by ×4.2. With 10 concurrent users, the increase of P90 is by at most ×2.3 w.r.t. 5 users. Differently than moving from 1 to 5 concurrent users, doubling the number of users doubles the response times in this case. This significant increase in latency is particularly evident for Q1, and it may underline as previously noted an overhead in query planning and execution for short queries. Such overhead may impact scaling, and needs to be investigated in future work. Nonetheless, these results show that Siren Federate scales well with the number of users and nodes, achieving subsecond to second response times over large datasets.

For Q4, Q5, and Q6, the aim is to further assess the ability of the system to scale with the amount of data, joining parent and child sets containing hundreds of millions to billions of documents each. With 36 nodes, the reported P90 is 1.92, 5.00, and 7.58 seconds for Q4, Q5, and Q6 respectively. We can observe a sub-linear scaling factor with the size of the join operation. Between Q4 and Q5 the size of the join operation increases by ×7, but P90 only increases by 2.2 seconds. Between Q5 and Q6, the size of the join operation increases by ×2 but P90 only increases by 1.5 seconds. These results confirm that Siren Federate scales well with the amount of data.

6 Conclusion

This paper presented the architecture of Siren Federate, a system that supports efficient exploratory analysis on large knowledge graphs by bridging document and relational data models. It addresses challenges in integrating graph analytics within a document-oriented IR system and demonstrates the effectiveness of this approach for supporting data-driven investigation systems.

Key contributions include integrating relational join operations within the document-oriented model, leveraging IR system capabilities, and implementing distributed join algorithms optimized for IR systems. We also introduced adaptive query planning for accurate runtime cardinality estimation, query plan folding to reduce redundant computations, and semantic caching to enhance iterative query performance. Columnar in-memory processing optimizations and Elasticsearch's log-structured distributed architecture were also highlighted.

Performance evaluations using a synthetic dataset with billions of documents demonstrated the efficiency and scalability of Siren Federate. The results show the capability to achieve sub-second to second response times on large datasets, which is critical for supporting the iterative workflow of investigators.

Our architecture retains the search and relevance ranking capabilities of the IR system while introducing efficient relational operations and graph analytics at scale. This demonstrates how a combination of document and relational system features can enhance scalability and analytical capabilities for exploratory analysis of large knowledge graphs.

Future efforts will focus on optimizing parallel processing of segments, incorporating spatial joins for enhanced analytics, expanding graph analytics, and developing new adaptive optimization techniques. Finally, we also plan to evaluate our system on additional, standard datasets, such as the LDBC Social Network Benchmark [2].

Acknowledgments. We used ChatGPT to improve writing in some parts of this otherwise original work. All AI-generated text has been carefully checked for correctness.

This version of the article has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-83472-1_13. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

References

1. Akhgar, B., Bayerl, P.S., Sampson, F. (eds.): Open Source Intelligence Investigation. Springer (2016)
2. Angles, R., Antal, J.B., Averbuch, A., Birler, A., Boncz, P., Búr, M., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Pey, J.L.L., Martínez, N., Marton, J., Paradies, M., Pham, M.D., Prat-Pérez, A., Püroja, D., Spasić, M., Steer, B.A., Szakállas, D., Szárnyas, G., Waudby, J., Wu, M., Zhang, Y.: The ldbc social network benchmark (2024)
3. Atzenbeck, C., Ozgul, F., Hicks, D.L.: Linking and organising information in law enforcement investigations. In: Proc. IV (2009)
4. Besta, M., Gerstenberger, R., Peter, E., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., Hoefler, T.: Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. ACM Comput. Surv. **56**(2) (2023)
5. Campinas, S., Catena, M., Delbru, R.: Method to Reduce Network Communication for Distributed Join on Shared-Nothing Databases. International (PCT) Application PCT/EP2023/059801 (2023)
6. Chasseur, C., Li, Y., Patel, J.M.: Enabling json document stores in relational systems. In: Proc. WebDB (2013)

7. Davoudian, A., Chen, L., Liu, M.: A survey on nosql stores. ACM Comput. Surv. **51**(2) (2018)
8. Delbru, R., Campinas, S.: Semantic caching of semi-join operators in shared-nothing and log-structured databases. U.S. Patent Application 17/597923 (2020)
9. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. FnT Databases **1**(1) (2007)
10. Gormley, C., Tong, Z.: Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine. O'Reilly (2015)
11. Hatcher, E., Gospodnetic, O.: Lucene in Action (In Action Series). Manning (2004)
12. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: A fast, efficient data structure for string keys. ACM Trans. Inf. Syst. **20**(2) (2002)
13. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G.D., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4) (2021)
14. Karpathiotakis, M., Alagiannis, I., Ailamaki, A.: Fast queries over heterogeneous data through engine customization. Proc. VLDB Endowment **9**(12) (2016)
15. Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., Boncz, P.: Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proc. VLDB Endowment **11**(13) (2018)
16. Leis, V., Radke, B., Gubichev, A., Kemper, A., Neumann, T.: Cardinality estimation done right: Index-based join sampling. In: Proc. CIDR (2017)
17. Lissandrini, M., Mottin, D., Hose, K., Pedersen, T.: Knowledge graph exploration systems: are we lost? In: Proc. CIDR (2022)
18. Liu, Z., Heer, J.: The effects of interactive latency on exploratory visual analysis. IEEE Trans. Vis. Comput. Graph. **20**(12) (2014)
19. Lu, A., Fang, Z.: Sql2fpga: Automatic acceleration of sql query processing on modern cpu-fpga platforms. In: Proc. FCCM (2023)
20. Lu, J., Holubová, I.: Multi-model databases: A new journey to handle the variety of data. ACM Comput. Surv. **52**(3) (2019)
21. Noel, S., Bodeau, D.J., McQuaid, R.: Big-data graph knowledge bases for cyber resilience. In: Proc. NATO IST-153 (2017)
22. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4) (1996)
23. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation for RDF data. In: Proc. ISWC (2006)
24. Papailiou, N., Tsoumakos, D., Karras, P., Koziris, N.: Graph-aware, workload-adaptive sparql query caching. In: Proc. SIGMOD (2015)
25. Pini, D., Tummarello, G., Delbru, R.: Optimization of database sequence of joins for reachability and shortest path determination. U.S. Patent 11720564 (2022)
26. Procopiuc, O., Agarwal, P.K., Arge, L., Vitter, J.S.: Bkd-tree: A dynamic scalable kd-tree. In: Advances in Spatial and Temporal Databases (2003)
27. Roijackers, J., Fletcher, G.H.L.: On bridging relational and document-centric data stores. In: Big Data (2013)
28. Sahal, R., Khafagy, M., Omara, F.: Big data multi-query optimisation with apache flink. Int. J. Web Eng. Technol. **13** (2018)
29. Sammons, J.: The basics of digital forensics: the primer for getting started in digital forensics. Syngress (2014)
30. Schuh, S., Chen, X., Dittrich, J.: An experimental comparison of thirteen relational equi-joins in main memory. In: Proc. SIGMOD (2016)
31. Ágnes Vathy-Fogarassy, Hugyák, T.: Uniform data access platform for sql and nosql database systems. Information Systems **69** (2017)

32. ten Wolde, D., Singh, T., Szárnyas, G., Boncz, P.: Duckpgq: Efficient property graph queries in an analytical rdbms. In: Proc. CIDR (2023)
33. Zhang, Z., Deshmukh, H., Patel, J.M.: Data partitioning for in-memory systems: Myths, challenges, and opportunities. In: Proc. CIDR (2019)