SEREIA - Document Store Exploration through Keywords

Ariel Antony Afonso Afonso

ariel.afonso@icomp.ufam.edu.br

Federal University of Amazonas

Paulo Rodrigo Oliveira Martins Martins

Federal University of Amazonas

Altigran Soares da Silva Soares

Federal University of Amazonas

Research Article

Keywords: information retrieval, data exploration, keywords, document stores

Posted Date: September 28th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3376351/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Knowledge and Information Systems on June 10th, 2024. See the published version at https://doi.org/10.1007/s10115-024-02151-1.

SEREIA - Document Store Exploration through Keywords

Ariel Afonso, Paulo Martins, Altigran da Silva

Institute of Computing, Universidade Federal do Amazonas, Avenida General Rodrigo Otávio, Manaus, 69067-005, Amazonas, Brazil.

Contributing authors: ariel.afonso@icomp.ufam.edu.br; paulo.rodrigo@icomp.ufam.edu.br; alti@icomp.ufam.edu.br;

Abstract

The adoption of *document stores*, such as MongoDB or CouchDB, has drastically increased in the past years. Part of this popularity can certainly be explained by their flexibility in terms of loading, storing, and retrieving semi-structured data on massive scales. However, adopting such systems presents challenges when exploring the data they store since the structure of the document may not follow a single pattern, and thus present complex hierarchical and nested structures that vary. Additionally, an analyst who wants to retrieve data may experience difficulties since she must learn the specificities of the document store's native query language. In this work, we propose *SEREIA*, a system that facilitates data exploration in document stores through keyword search. The user inputs a non-structured keyword-based query and the system generates a structured query for the document store that fulfills her information needs. We evaluated *SEREIA* using five datasets previously used in the literature and the results we achieved indicate that *SEREIA* is suitable for helping users in data exploration tasks by removing the burden of understanding the data organization of the stored documents and by automatically generating queries to explore data of interest.

Keywords: information retrieval, data exploration, keywords, document stores

1 Introduction

The adoption of *document stores*, such as MongoDB or CouchDB, has increased drastically in the past years. Currently, this kind of system is adopted in several domains and many applications [1]. Part of this popularity is certainly due to their flexibility in terms of loading, storing, and retrieving semi-structured data on massive scales [2, 3]. This flexibility is mainly due to two aspects. First, document stores allow the creation of semi-structured documents

with complex hierarchical and nested structures [2]. Second, they do not require a schema definition [4]. In fact, they support the so-called *schema later* approach, i.e., they allow "relaxed" data ingestion, without requiring that documents follow a predefined structure [3].

Dealing with these aspects adds a degree of complexity to the stored data and incurs challenges that are noticed when data scientists need to retrieve information from these systems [5–7]. Indeed, due to their lack of strict structure declarations, document stores require the user to have a deep understanding of the underlying structure and organization of the data when seeking information. In addition, another challenge is that the query languages used by document stores are considerably more complex than, for instance, SQL. In fact, document stores challenge even experienced data scientists who want to analyze the data they store [8], resulting in extra time and effort when looking for relevant data, rather than effectively analyzing it [6].

Previous works in the literature have presented proposals to facilitate the retrieval of data from document stores. Systems such as *Argo* [4] and *Sinew* [3] employ an intermediate layer that maps documents to relational structures, thus allowing the use of SQL in collections of documents. In addition, Liu et al. [9] proposed an approach for storing documents in relational databases that consequently allows the use of SQL on documents without requiring the mapping step. Although these approaches facilitate the analysis of data in documents by leveraging SQL, the user must still have knowledge of the structure of the document collection.

In this work, we propose *SEREIA*, a system that facilitates data exploration in document stores. More specifically, *SEREIA* takes as input a non-structured keyword-based query and generates a corresponding structured query written in the document stores' native query language. By doing so, we aim to help data analysts/scientists undertake their data analysis tasks without having to deal with the typical complexities of retrieving data from document stores. In particular, *SEREIA* avoids the need for users to have prior knowledge of the underlying structure and organization of the stored documents, besides the syntax and operations of its native query language. As we show through experiments, *SEREIA* allows one to explore the contents of documents using a handful of keywords that express their information needs.

Our work with *SEREIA* is in line with previous work that has been developed to help explore semi-structured or unstructured data. This includes interfaces that allow data discovery by example [10] and using intermediary structures, e.g., knowledge graphs, that yield issuing queries at a higher abstraction level [6, 11]. Although these proposals provide higher-level interfaces for data exploration, they still require previous knowledge of the schema and the adoption of some query language, such as SPARQL.

Our work extends previous work on keyword queries in databases, specifically *MatCNGen* and *CNRank* [12–14]. To the best of our knowledge, *SEREIA* is the first system to propose exploring document stores directly using keyword-based search. Our system maps each keyword to an attribute in documents of a collection, based either on the name of the attribute or on its values. Then, the system combines these mappings, generating possible interpretations of the keyword query provided. A ranking algorithm is used to score the combinations that are more likely to satisfy the user's intent. From these higher-scored combinations, *SEREIA* looks for ways to aggregate each mapping presented in a combination. This result is used to generate a structured query in the document store's native language that fulfills the original keyword query.

To evaluate *SEREIA*, we performed experiments using MongoDB[1], currently the most popular document store [2]. We used five datasets, of which four were used in previous work. For each dataset, we present a set of keyword queries, analyze their results, and analyze the way *SEREIA* generates them, using information retrieval metrics. We also measure the time it takes the system to generate and execute the query on the document store.

This paper is organized as follows: in Section 2 we discuss related work. In Section 3 we present an overview of the architecture of *SEREIA* and illustrate its functioning using an example. In Sections 4, 5 and 6 we present and discuss the techniques that *SEREIA* employ to generate a fully structured query given a keyword query. In Section 7 we illustrate how the structured query is generated, along with the algorithm for this process. In Section 8 we present experimental results. Finally, in Section 9 we discuss next steps regarding *SEREIA* and summarize our findings.

2 Related Work

Previous works have presented proposals to facilitate retrieving data from document stores, leveraging users' knowledge in SQL, introducing intermediary layers to avoid dealing with semi-structured collections of documents, and mapping documents to relational structures. Systems such as *Argo* [4] and *Sinew* [3] materialize data from collections of documents in relational tables in a preprocessing step, thus allowing the data to be retrieved later using SQL. In addition, Liu et al. [9] propose storing documents in a relational structure without previously mapping the document's fields, thereby eliminating the preprocessing step. However, all the approaches that adopt SQL to facilitate the retrieval of data from document stores still challenge the user to comprehend the underlying structure and organization of the data.

To overcome such a problem, our system adopts a simpler approach, i.e., it provides a keyword search interface over document stores. For the past two decades, researchers have extensively studied relational keyword search systems (R-KwS) [15]. In general, these systems fall into two categories. The first category represents *schema-based* systems whose objective is to generate SQL queries given a keyword query by building a graph, called *Schema Graph*. In this graph, the nodes represent relations and the edges represent referential integrity constraints. From this graph, many *Candidate Joining Networks* (CJNs)[3] can be generated. A CJN is a joining tree that represents an interpretation of the original keyword query and describes how the data in the underlying database can be joined to achieve the expected result for a given keyword query. For every CJN, an SQL query may be generated. State-of-the-art systems in this category employ an early evaluation to assess which CJNs better represent the original keyword query. This evaluation improves not only the overall quality of the process, but also its efficiency [12–14, 16]. The second category represents *instance-based* systems that aim to materialize database tuples in a graph called *Data Graph*. In this graph, the nodes represent the tuples of the database containing keywords, and the edges represent the constraints of the referential integrity. Following this approach, the answer is a subtree of the data graph that minimizes the distance between nodes containing the keyword query terms.

Our system, *SEREIA*, is based on the MatCNGen system [13, 14], which falls into the first category. Natively, document stores support nested documents, and avoid references between data items, which are common in relational databases. However, this increases the complexity

[1] https://www.mongodb.com/
[2] https://db-engines.com/en/ranking/document+store
[3] These structures were originally called Candidate Networks [15]

of generating queries in document collections that correctly capture the user's intent, as the data sought can be found in a single collection or spanned in multiple collections, which may or not be joinable according to the schema. However, by adapting the concept of CJNs, *SEREIA* generates query alternatives that help the user quickly explore a document store.

Recently, Natural Language Interfaces for Databases (NLIDBs) have aroused interest. Systems such as NaLIR [17] focus on mapping query terms to database schema elements using techniques such as syntactic dependency rules and ontologies. On the other hand, systems such as SQLizer [18] focus on translating NL queries into SQL using deep learning techniques. Keyword-based query systems offer distinct advantages over Natural Language Interface to Database (NLIDB) systems, particularly in the realm of data exploration and query efficiency. Unlike NLIDBs, which often necessitate detailed and specific user input to generate precise answers, keyword-based systems can provide a comprehensive range of information with simple keyword queries. For instance, in a database containing data on National Basketball Association (NBA) games, a query such as "Stephen Curry" would yield all pertinent information about the player without requiring a complete sentence or specific question. These queries are often categorized as "navigational queries," a term that is well-established in the field of information retrieval. Such queries are advantageous for users interested in exploring the database's content without a predefined question. The simplicity of keyword-based systems also leads to faster query execution, as they bypass the computational overhead associated with natural language processing. Moreover, past research [19] has indicated that NLIDBs may overlook or misinterpret valuable keywords in the query, thereby affecting the accuracy of the results. In contrast, keyword-based systems utilize every term in the query, ensuring that all relevant information is considered. In summary, keyword-based query systems offer a more straightforward, efficient, and often more accurate method for database exploration compared to NLIDBs.

Our work is also in line with previous work developed to help users explore semi-structured or unstructured data. Rezig et al. [10] introduced higher-level interfaces that allow the user to provide examples that are used to retrieve similar data. The main goal is to synthesize an SQL query that correctly captures the user intent through the provided examples. Other works provide a more standard way of extracting data from the underlying data sources using knowledge graphs, which can be queried with SPARQL or proposed query languages [6, 11]. Although these proposals aim at higher-level interfaces that facilitate data retrieval, they require the user to perform several queries to find relevant data and join paths. In contrast, *SEREIA* generates alternative queries that are likely to correspond to the intent of the user, ranking the ones that most likely suit the needs of the user.

3 Overview

In this section, we present an overview of *SEREIA*. We base our discussion on a simplified excerpt of the *Yelp!* database, illustrated in Figure 1, and which identifies documents by an ID in the top right corner and, also, illustrates how documents connect to each other.

Consider that a user inputs the keyword query "*italian restaurants reviewed michelle*", expecting to retrieve all documents containing data on italian restaurants that were reviewed by user Michelle. The result that satisfies this keyword query is obtained by joining data from documents $D2$, $D6$ and $D10$ from the excerpt, resulting in the output shown in Figure 3. To retrieve results satisfying the keyword query's intent, one must issue a structured query to the

underlying document store. A possible structured query that correctly retrieves the expected result is shown in Figure 2.

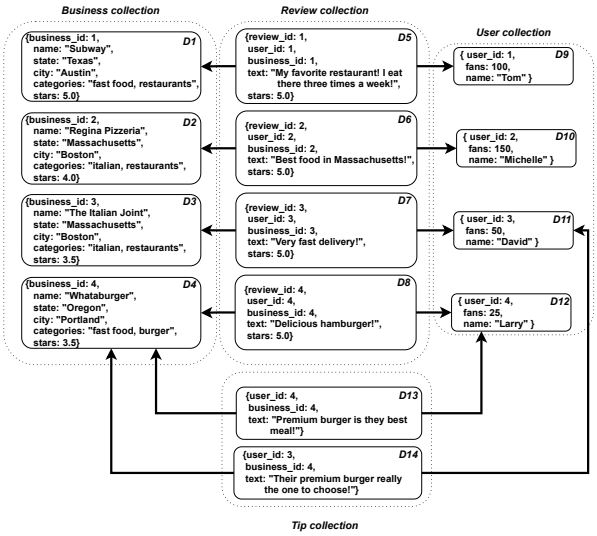


Fig. 1: An excerpt from collections in the *Yelp!* database

```
db.user.aggregate([
{ "$match": { "$expr": { "$regexMatch": {
    "input": "$user.name", "regex": "michelle",
    "options": "i"}}}},
{ "$lookup": { "from": "review",
  "foreignField": "user_id",
  "localField": "user_id", "as": "review"}},
{ "$unwind": "$review" },
{ "$lookup": { "from": "business",
  "foreignField": "business_id",
  "localField": "review.business_id", "as": "business"}},
{ "$unwind": "$business" },
{ "$match": { "$expr": { "$regexMatch": {
    "input": "$categories", "regex": "restaurants",
    "options": "i"}}},
{ "$match": { "$expr": { "$regexMatch": {
    "input": "$categories", "regex": "italian",
    "options": "i"}}}}])
```

Fig. 2: MongoDB Structured Query

SEREIA Architecture

We note that manually crafting such a query requires expertise in the query language of the document store and, thus, a casual non-expert will find it difficult to explore documents in a database of interest. This worsens if we consider that the database may have an intricate structure with different collections with interconnected and nested documents.

Next, we provide an overview of how *SEREIA* can be used to achieve the same results in a more intuitive and easy way that allows casual users to explore document databases using simpler unstructured keyword queries.

```
{ "business_id": 2,
  "name": "Regina Pizzeria",
  "state": "Massachusetts",
  "categories": "italian, restaurants",
  "stars": 4.0,
  "review": {
    "review_id": 2, "user_id": 2,
    "business_id": 1, "stars": 5.0,
    "text": "Best food in Massachusetts!" },
  "user": {
    "user_id": 2, "name": "Michelle",
    "fans": 150 }}}
```

Fig. 3: Output for the keyword query

In Figure 4, we illustrate *SEREIA*'s architecture. This architecture relies on some techniques previously presented in the literature [12, 13, 16] that allow keyword-based queries in relational databases, which in our work we adapted to the context of document stores. Thus, we use a terminology similar to the one adopted in previous work.

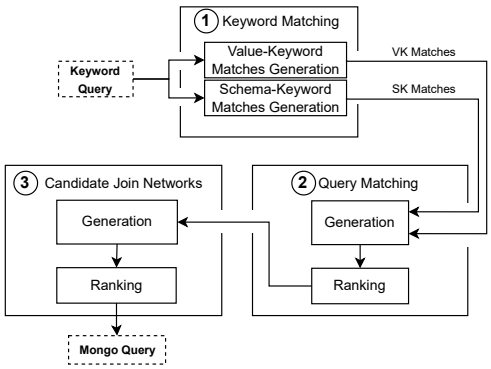


Fig. 4: *SEREIA* Architecture

The process begins when a user issues a keyword query. Then, our system attempts to associate each keyword from the query with the attributes of a collection's document. This association can be either based on the attribute's name or on its values. We call this phase *Keyword Matching* (1), whereby *SEREIA* associates keywords with attributes based on the values containing these keywords, generating *Value-Keyword Matches* (VKMs), or associates keywords with attribute or collection names, generating *Schema-Keyword Matches* (SKMs).

In Table 1 we show possible matches between keywords in the input query and document attributes or database elements. For example, the term "*italian*" is likely to refer to information in the `Business` collection, specifically in attributes `name` or `categories`, as illustrated by documents $D2$ and $D3$ in Figure 1. Similarly, the term "*restaurants*" is likely to refer to information in the attribute `categories`, from the `Business` collection or in the attribute `text`, from the `Review` collection, as illustrated by documents $D1$, $D2$, $D3$ and $D5$ in Figure 1. Since these keywords are part of attribute values, these matches are considered

VKMs. In the case of the keyword "*reviewed*", it actually matches the *name* of the `Review` collection, which is why in Table 1 the keyword "*reviewed*" matches `Review.self`. Thus, this match is considered an SKM. The *Keyword Matching* phase is detailed in Section 4.

Table 1: Query keywords matched to document attributes

Keyword	Matching Attribute
italian	`Business.categories`, `Business.name`
restaurants	`Business.categories`, `Reviews.text`
reviewed	`Review.self`
michelle	`User.name`

In sequence, *SEREIA* generates combinations of VKMs and SKMs. In these combinations, we consider that all keywords in the query must be matched; in other words, the combination must be *total*. Furthermore, we also consider that all pairs of keywords and attributes are "useful"; that is, if we remove any of the pairs, this would result in a non-total combination. All combinations that satisfy both criteria are called *Query Matches* (QMs) (2). In Figure 5 we present all possible QMs of the KMs illustrated in Table 1.

italian→`Business.categories`
restaurants→`Business.categories`
reviewed→`Review.self`
michele→`User.name`
(a)

italian→`Business.name`
restaurants→`Review.text`
reviewed→`Review.self`
michele→`User.name`
(b)

italian→`Business.name`
restaurants→`Business.categories`
reviewed→`Review.self`
michele→`User.name`
(c)

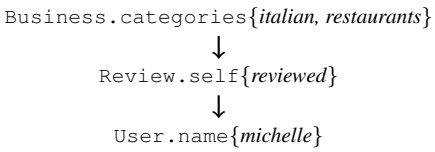
italian→`Business.categories`
restaurants→`Review.text`
reviewed→`Review.self`
michele→`User.name`
(d)

Fig. 5: Examples of combinations of keywords matched.

Although many combinations are possible, it can be noted that the only combination that fulfills the intention of retrieving documents containing data on Italian restaurants that were reviewed by Michelle is the combination in Figure 5 (a). In general, just a few combinations correspond to the user's intention. Thus, to select such combinations, *SEREIA* uses a ranking algorithm so that only the best ranked QMs are further processed. We present the details on QMs, their generation, and ranking in Section 5.

The QMs described above identify which collections and documents contain the information sought by the user when formulating a query. However, to produce a proper answer that integrates this information, it is also necessary to properly "aggregate" documents necessary to answer the query. To achieve this, *SEREIA* defines a sequence of processing stages on the data, resulting in structures called *Pipeline Sketches* (PLSs). Thus, PLSs describes how data from the document store collections can be processed to produce an answer to the keyword query. The information from the QM collections shown in Figure 5 (a) can be integrated as follows:

where arrows indicate references between documents in the collections. It denotes that we look for documents from the `Business` collection whose values of the `categories`



attribute contain the terms "*italian*" and "*restaurants*", which are referred to by documents in the `Review` collection, which, in turn, also refer to documents in the `User` collection whose attribute `name` contains the term "*michelle*". As the figure suggests, PLSs are paths, and we will define their structure more precisely in Section 6.

As described above, *SEREIA* generates PLSs from QMs. Similarly to the Query Matching phase, we also developed a PLS ranking algorithm to aid in selecting the best PLS that fulfills the user's keyword query intent. Then, *SEREIA* translates this PLS into a query that retrieves data from the underlying document store. Considering the combination in Figure 5 (a), *SEREIA* generates the structured query shown in Figure 2. In Section 7 we detail our algorithm for generating a structured query from a given PLS.

During the PLS generation phase, *SEREIA* uses two data structures generated in a preprocessing phase (0): the Value Index and the Schema Index. The Value Index is an inverted index that stores keywords and their occurrences in the document store, indicating their frequency and indicating the collections and document attributes they were located. This information is used when generating VKMs. The Schema Index is an inverted index that stores information about collections and document attributes, and also statistics such as *norm* and *inverted frequency* useful when ranking QMs and PLSs. This index is used when generating SKMs.

In the next sections, we present an in-depth explanation of *SEREIA* structure, processing phases, and algorithms. *SEREIA* was fully implemented and is available at https://github.com/bdri-ufam/sereia.

4 Keyword Matching

During the Keyword Matching phase, *SEREIA* can associate keywords from the query with document attributes based on their values, using *value-keyword matches*.

Definition 1. *Consider a keyword query Q, a collection of documents C, and a set $\mathcal{A} = \{A_1, \ldots, A_n\}$ of attributes from all documents in C. A* ***value-keyword match*** *is a subset of documents of C in which all documents contain a subset of the terms of Q in at least one of its attributes. In other words, a VKM is a set of documents*

$$C^V[A_1^{K_1}, \ldots, A_n^{K_n}] = \{d | d \in C \wedge \forall A_i : W(d[A_i]) \cap Q = K_i\}$$

where $W(d[A_i])$ returns the set of keywords found in the attribute A_i for document d and K_i is the subset of keywords from Q associated with the attribute A_i.

Given a keyword query $Q_1 =$ "*italian restaurants reviewed michelle*" and the collections of documents illustrated in Figure 1, possible value-keyword matches are:

$$\texttt{User}^V[name^{\{michelle\}}] = \{D8\}$$
$$\texttt{Business}^V[categories^{\{italian\}}] = \{D2, D3\}$$
$$\texttt{Business}^V[categories^{\{restaurants\}}] = \{D1, D2, D3\}$$

In the above VKM, we illustrate how *SEREIA* relates keywords, attributes, and documents. Consider the VKM $\texttt{User}^V[name^{\{michelle\}}]$. For instance, we indicate the match between the keyword "*michelle*" and the `name` attribute from the `User` collection. In this case, the match refers to document $D8$ from Figure 1. *SEREIA* is also capable of associating keywords to collection or attribute names using *schema-keyword matches*.

Definition 2. *Consider a keyword query Q, a collection of documents C and a set of attributes of all documents in C to be $\mathcal{A} = \{A_1, \ldots, A_m\}$. A **schema-keyword match** is a subset of documents of C in which at least one attribute name corresponds to a keyword in the query. That is, an SKM is a set of documents*

$$C^S[A_1^{K_1}, \ldots, A_m^{K_m}] = \{d \mid d \in C \wedge \forall k \in K_i : match(k, A_i)\}$$

where $1 \leq i \leq m$, K_i is the subset of keywords from Q that are associated to A_i, match(A_i, K_i) estimates the similarity between the schema element A_i and the keyword k, and S indicates a match of keywords to the database schema.

Given the keyword queries Q_2 = "*cities subway*", Q_3 = "*states Whataburguer*" and the collections of documents illustrated in Figure 1, possible schema-keyword matches are:

$$\texttt{Business}^S[state^{\{states\}}] = \{D1, D2, D3, D4\}$$
$$\texttt{Business}^S[city^{\{cities\}}] = \{D1, D2, D3, D4\}$$

Notice that, differently from VKMs, SKMs refer to attributes found across the entire collection. That is, SKMs do not "filter" documents based on attribute values, but rather indicate documents that contain an attribute that matches a keyword from the query.

Finally, in some cases, a keyword in the query can refer to a whole collection, as in the case of the term "*reviewed*" from the query Q_1. To address this, we extend the schema-keyword match notation to include a specific attribute to represent a collection. We call this attribute $self$ and consider $match(self, k)$ to be true if k corresponds to a collection's name. Regarding this scenario, the schema-keyword match for the term "*reviewed*" from Q_1 is defined as:

$$\texttt{Review}^S[self^{\{reviewed\}}] = \{D5, D6, D7, D8\}$$

The similarity function *match* used above can be implemented in many different ways. Currently, this function matches a keyword and a schema element by applying a lemmatizer and a stemmer, then executes an exact match on the processed results.

In the cases in which there are SKMs and VKMs for a query in the same collection, it is useful to represent them using a single expression, which we call a *Keyword Match* (KM).

Definition 3. *Consider a keyword query Q, a collection of documents C and a set of attributes from all documents in C to be $\mathcal{A} = \{A_1, \ldots, A_m\}$. Let $VKM = C^V[A_1^{K_1^V}, \ldots, A_m^{K_m^V}]$ be a value-keyword match from C in Q. Let $SKM = C^S[A_1^{K_1^S}, \ldots, A_m^{K_m^S}]$ be a schema-keyword match from C in Q. A general **keyword match** from C in Q is given by:*

$$C^S[A_1^{K_1^S}, \ldots, A_m^{K_m^S}]^V[A_1^{K_1^V}, \ldots, A_m^{K_m^V}] = VKM \cap SKM$$

We note that a KM must satisfy both its components VKM and SKM. Thus, the documents in the KM are those in VKM $\cap$ SKM.

5 Query Matching

In general, many combinations of KMs of a keyword query may return documents that, when correctly connected, satisfy the query given as input. However, in accordance with previous work [12, 13, 15, 16], we consider in this work only combinations that correspond to a "total" and "minimal" cover of a keyword query. More precisely, a total cover guarantees that all terms from the keyword query are present in the result, and a minimal cover guarantees that there are no documents with redundant information. Combinations that satisfy both criteria are called *query matches* as formally defined below.

Definition 4. *Let Q be a keyword query. Let $M = \{KM_1, \ldots, KM_n\}$ be a set of keyword matches for Q in a certain collection C, where:*

$$KM_i = C_i^S[A_{i,1}^{K_{i,1}^S}, \ldots, A_{i,m_i}^{K_{i,m_i}^S}]^V[A_{i,1}^{K_{i,1}^V}, \ldots, A_{i,m_i}^{K_{i,m_i}^V}]$$

*Also consider $\mathcal{K}_{M_i} = \bigcup_{\substack{1 \leq j \leq m_i \\ X \in \{S,V\}}} K_{i,j}^X$, the set of all keywords associated with KM_i and $\mathcal{K}_M = \bigcup_{1 \leq i \leq n} \mathcal{K}_{M_i}$, be all keywords associated and with M. We say that M is a **query match** for Q if, and only if, $\mathcal{K}_M$ forms a **minimal set cover** of the keywords in Q, that is, $\mathcal{K}_M = Q$ and $\mathcal{K}_M \backslash \mathcal{K}_{M_i} \neq Q$, $\forall \mathcal{K}_{M_i} \in M$.*

Given the keyword query $Q =$"*italian restaurants business reviewed michelle*", possible VKMs and SKMs combinations are:

$$\begin{aligned}
M_1 = \{&\texttt{User}^V[name^{\{michelle\}}], \texttt{Business}^S[self^{\{business\}}], \\
&\texttt{Business}^V[categories^{\{restaurants,italian\}}], \texttt{Review}^S[self^{\{reviewed\}}]\} \\
M_2 = \{&\texttt{User}^V[name^{\{michelle\}}], \\
&\texttt{Business}^V[name^{\{italian\}}], \texttt{Review}^S[self^{\{reviewed\},}]\} \\
M_3 = \{&\texttt{User}^V[name^{\{michelle\}}], \texttt{Business}^V[categories^{\{restaurants,italian\}}], \\
&\texttt{Business}^V[name^{\{italian\}}], \texttt{Review}^S[self^{\{reviewed\}}]\}
\end{aligned}$$

From these combinations, only M_1 is considered a Query Match. M_2 does not contain the term "*restaurants*" from the keyword query Q, thus it is not total. On the other hand, M_3 is redundant, since removing the VKM $\texttt{Business}^V[name^{\{italian\}}]$ would still result in a total combination. Thus, M_3 is not minimal.

Query Match Ranking

Since many QMs can be generated for a single keyword query, *SEREIA* uses a ranking algorithm based on the CNRank algorithm [12] to identify QMs that are useful and more prone to satisfy the user's intent. *SEREIA* estimates the relevance of QMs based on a *Bayesian Belief Network* model for the current state of the underlying database [16]. In practice, this model evaluates two types of relevance evidence when ranking QMs. The TF-IDF model is used to calculate a *value-based score*, which adapts the traditional Vector space model [20] to the context of relational databases, as occurs in LABRADOR [21] and CNRank [12]. On the other hand, the *schema-based score* is calculated by estimating the similarity between keywords and schema element names. In *SEREIA*, only the top-k QMs in the ranking are considered in following phases. By doing so, we avoid generating PLSs that are less likely to correctly interpret the keyword query.

To emphasize the importance of ranking QMs in *SEREIA*, we found out in our experiments that certain keyword queries may generate hundreds or even thousands of QMs. This is due to the high frequency of some keywords in several documents. For instance, in the *IMDb* database, the average number of QMs per query is above 2500. The ranking process allows *SEREIA* to discard most of these spurious QMs and focus on a handful of QMs that, as we show experimentally, are likely to include the one that correctly satisfies the user's needs.

6 Pipeline Sketches Generation and Ranking

To generate PLSs, *SEREIA* uses an auxiliary structure called *Schema Graph*. In this graph, every node represents a collection from the document store, while the edges represent references between documents of distinct collections.

Definition 5. *Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of collections from a document store. Let E be a subset of the ordered pairs from $\mathcal{C}^2$ given by $E = \{\langle C_a, C_b\rangle | \langle C_a, C_b\rangle \in \mathcal{C}^2 \wedge C_a \neq C_b \wedge REF(C_a, C_b) \geq 1\}$, where $REF(C_a, C_b)$ gives the number of references from documents of a collection C_a to documents of a collection C_b. We say that a* ***schema graph*** *is an ordered pair $G_S = \langle \mathcal{C}, E\rangle$, where $\mathcal{C}$ is the set of vertices (nodes) of G_S, and E is the set of edges of G_S.*

The function *REF* in Definition 5 is a way of indicating relationships between different collections in the target document store. For instance, in Figure 1 we show documents from the `REVIEW` collection that "link" to documents from the collections `USER` and `BUSINESS`. This is reflected in the schema graph. Notice that the relationships between documents may also be implemented by nesting all documents that represent other related entities in the structure of a document. However, we do not consider this kind of relationship in our work. Considering the excerpt in Figure 1, *SEREIA* generates the schema graph below:

$$G_S = \begin{array}{ccc} \text{User} & \leftarrow & \text{Review} \\ \uparrow & & \downarrow \\ \text{Tip} & \rightarrow & \text{Business} \end{array}$$

6.1 Pipeline Sketch Generation

After defining the concept of a schema graph, we introduce the concept of *Pipeline Sketch* (PLS). Roughly speaking, a PLS is a graph whose nodes include all KMs from a QM and follows constraints that assure the generation of a structured query that is valid and properly retrieves data from the document store.

Consider the keyword query Q_1 = "*businesses bricola stars 5.0*" for the *Yelp!* database, which, intuitively aims at retrieving the businesses which user "Bricola" rated with 5 stars. One of the query matches generated for this keyword query is the following:

$$\begin{aligned} QM_6 = \{&\texttt{User}^V[\texttt{name}^{\{bricola\}}], \\ &\texttt{Review}^S[\texttt{stars}^{\{stars\}}]^V[\texttt{stars}^{\{5.0\}}], \\ &\texttt{Business}^S[\texttt{self}^{\{businesses\}}]\} \end{aligned}$$

The following PLS can be generated for QM_6:

$$\begin{array}{c} P_1 = \texttt{User}^V[\texttt{name}^{\{bricola\}}] \\ \downarrow \\ \texttt{Review}^S[\texttt{stars}^{\{stars\}}]^V[\texttt{stars}^{\{5.0\}}] \\ \downarrow \\ \texttt{Business}^S[\texttt{self}^{\{businesses\}}] \end{array}$$

The PLS above illustrates how the KMs from a QM can be connected to retrieve information from the document store. In the provided example, *SEREIA* uses the information from the schema graph to generate a pipeline of stages between all the collections specified by QM_6, aiming at retrieving data from the document store. In addition, there can be cases in which the collections from the KMs are not adjacent to each other in the graph and, thus, it is necessary to include intermediate collections to correctly retrieve an answer the provided keyword query.

Consider again the keyword query $Q =$ "*businesses bricola stars 5.0*" for the *Yelp!* database. Another possible query match for this query is the following:

$$QM_7 = \{\texttt{User}^V[\texttt{name}^{\{bricola\}}],\\ \texttt{Business}^S[\texttt{self}^{\{businesses\}}]^V[\texttt{name}^{\{stars\}}, \texttt{stars}^{\{5.0\}}]\}$$

For this query match, two different PLSs can be generated, given the schema graph generated for the *Yelp!* database:

$$P_2 = \texttt{User}^V[\texttt{name}^{\{bricola\}}]$$
$$\downarrow$$
$$\texttt{Review}$$
$$\downarrow$$
$$\texttt{Business}^S[\texttt{self}^{\{businesses\}}]^V[\texttt{name}^{\{stars\}}]$$
$$\text{- -}$$
$$P_3 = \texttt{User}^V[\texttt{name}^{\{bricola\}}]$$
$$\downarrow$$
$$\texttt{Tip}$$
$$\downarrow$$
$$\texttt{Business}^S[\texttt{self}^{\{businesses\}}]^V[\texttt{name}^{\{stars\}}]$$

The PLSs above contain additional nodes that are used to connect non-adjacent KMs. Considering PLS P_1, notice that, in order to aggregate data from the *User* and *Business* collections, *SEREIA* uses the *Review* collection as an intermediate collection to aggregate all the collections required by QM_7. In that example, *SEREIA* has more than one possibility for aggregating both collections. The first option is using `Review` as an intermediate collection, as seen in P_2, and the second one is using the `Tip` collection, as seen in P_3. In both cases, the intermediate collection is adjacent to the collections involved in the query match according to the schema graph.

In our work, based on the terminology used in previous work [13, 16], we call these intermediate collections *keyword-free matches*, which are essential for the definition and generation of Pipelines Sketches. Specifically, these matches assist when generating PLSs by acting as intermediate nodes to other KMs.

Definition 6. *We say that a keyword match* $KM = C^S[A_1^{K_1^S}, \ldots, A_m^{K_m^S}]^V[A_1^{K_1^V}, \ldots, A_m^{K_m^V}]$ *is a* ***keyword-free match*** *if, and only if,* $\nexists K_i^S \neq \{\} \wedge \nexists K_i^V \neq \{\}$*, where* $1 \leq i \leq m$*.*

For a more concise notation, we represent a keyword-free match as $C^S[\,]^V[\,]$ or simply C. For the sake of example, given the keyword match $\texttt{User}^S[]^V[]$, we represent it as `User`, as presented in a schema graph. Now, we formally define the concept of PLSs.

Definition 7. *Let Q be a keyword query over a document store DS, whose schema graph is G_S. Also, let M be a query match for Q in DS. Let F be the set of keyword-free matches from collections of G_S. Consider a path graph of keyword matches $P = \langle \mathcal{V}, E \rangle$, where $\mathcal{V}$ and E*

are the vertices and edges of J. We say that P is a ***pipeline sketch*** *from M over G_S if:*

$$(i)\mathcal{V} = M \cup F$$
$$(ii)\forall\langle KM_a, KM_b\rangle \in E \implies \exists\langle KM_a, KM_b\rangle \in G_S$$
$$(iii)\forall KM_a \in V_P : \forall\langle C_a, C_b\rangle \in E_G :$$
$$REF(C_a, C_b) \geq |\{KM_c | \langle KM_a, KM_c\rangle \in E_P \wedge C_c = C_b\}|$$

In Definition 7, we present the constraints used by *SEREIA* to generate PLSs. According to (i) a PLS can be formed by keyword matches and keyword-free matches, in the case that some KMs from a QM are not adjacent to each other. In (ii), we define that the edges from the PLS follow the structure of the schema graph. Finally, (iii) defines the concept of *soundness*. A sound PLS indicates that a KM cannot point to the same collection more times than the number of attributes that reference that same collection. This concept of soundness is adapted from previous work [16].

Consider the query match QM_8:

$$QM_8 = \{\texttt{User}^V[\texttt{average\_stars}^{\{5.0\}}], \texttt{User}^V[\texttt{name}^{\{bricola\}}],$$
$$\texttt{Business}^S[\texttt{self}^{\{businesses\}}]^V[\texttt{name}^{\{stars\}}]\}$$

The following structure can be generated for QM_8:

$$P_4 = \texttt{User}^V[\texttt{name}^{\{bricola\}}]$$
$$\downarrow$$
$$\texttt{Review}$$
$$\downarrow$$
$$\texttt{Business}^S[\texttt{self}^{\{businesses\}}]^V[\texttt{name}^{\{stars\}}]$$
$$\downarrow$$
$$\texttt{User}^V[\texttt{average\_stars}^{\{5.0\}}]$$

Recalling the definition of VKMs, $\texttt{User}^V[\texttt{name}^{\{bricola\}}]$ and $\texttt{User}^V[\texttt{average\_stars}^{\{5.0\}}]$ are disjoint sets of documents. By inspecting Figure 1, which shows an excerpt of the *Yelp!* database, we can safely assume that the *Review* collection presents documents that refer only to a single user. Thus, a document from `Review` cannot refer to two distinct users.

Given P_4, this pipeline sketch would generate a query whose result is empty, as there is no user with the name *bricola* and that presents an average star rating of 5. Per Definition 7, P_4 presents an inconsistency and is considered an unsound PLS. From past examples, only P_1, P_2, and P_3 are considered valid PLSs.

6.2 Pipeline Sketch Ranking

As shown for QM_7, a single QM may generate more than one PLS, resulting in a large number of PLSs when considering several QMs. To prune spurious PLSs, which often do not provide a plausible answer to the keyword query or do not present an answer at all, we adopt a ranking algorithm that uses the previously computed QM scores while also penalizing large PLSs, by decreasing the score proportionally to the size of the PLS. Therefore, the score of a pipeline sketch P_M from a query match M is given by:

$$score(P_M) = score(M) \times \frac{1}{|P_M|}$$

Consider PLSs P_1, P_2 and P_3 from previous examples. Our ranking algorithm outputs them in the following order: P_1, P_2 and P_3. Regarding P_2 and P_3, these PLSs present the same score since they indicate the same collections, just differing in the keyword-free match that joins the desired collections. As keyword-free matches are not scored because they do not match attributes, they do not affect the PLS score. In addition, these PLSs present a higher number of KMs, being more penalized. As for P_1, this PLS is more concise than the others in the number of KMs and presents higher overall scores considering the individual KMs score.

7 Structured Query Generation

To provide users with an answer to their keyword queries, *SEREIA* generates a structured query corresponding to one of the best-ranked PLSs. As mentioned, in this work we adopted MongoDB as the underlying document store. In many cases, queries in MongoDB document collections require using *Aggregation Framework Pipeline*[4]. This pipeline is based on different *stages* that receive a collection as input and process it by: (i) adding new attributes to the collection's documents; (ii) modifying existing attributes, or (iii) filtering out documents that do not satisfy a given condition. Each stage takes as input the data produced as the output from the previous one. To properly present our algorithm for structured query generation, we first need to overview the types of MongoDB stages we use and how they process the input data. This is discussed next.

7.1 Query Stages in MongoDB

We briefly discuss the types of MongoDB stages we used in our work by means of a running example, that illustrates how these stages modify the input data. Notice that all the semantics presented here are inherent to MongoDB's Aggregation Framework.

`$match` Stage

Given a term from the keyword query located in an attribute's value, *SEREIA* generates a `$match` stage to allow only documents that present a given value in that attribute, as seen in Figure 6. This stage searches for values using regular expressions in the documents attributes.

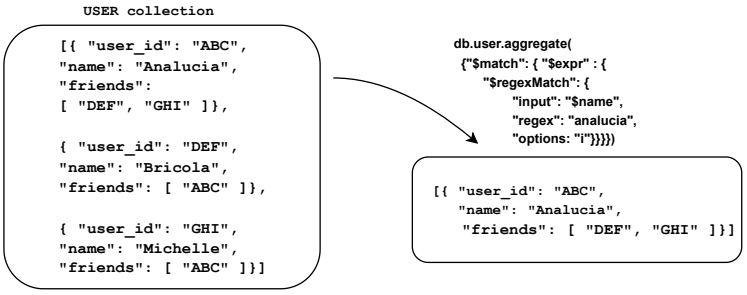


Fig. 6: $match query stage and output example

`$lookup` Stage

Considering scenarios in which a document database presents more than one collection, such as the *Yelp!* database, MongoDB allows the join operation through the `$lookup` stage, as seen in Figure 9. Essentially, this operator simulates the `LEFT JOIN` operation known in

[4] https://docs.mongodb.com/manual/aggregation/

relational databases. By default, in this work, the `$lookup` stage always generates a new attribute in the document with the same name as the collection. The result of satisfying all conditions in the `$lookup` stage is added as an attribute to the base collection.

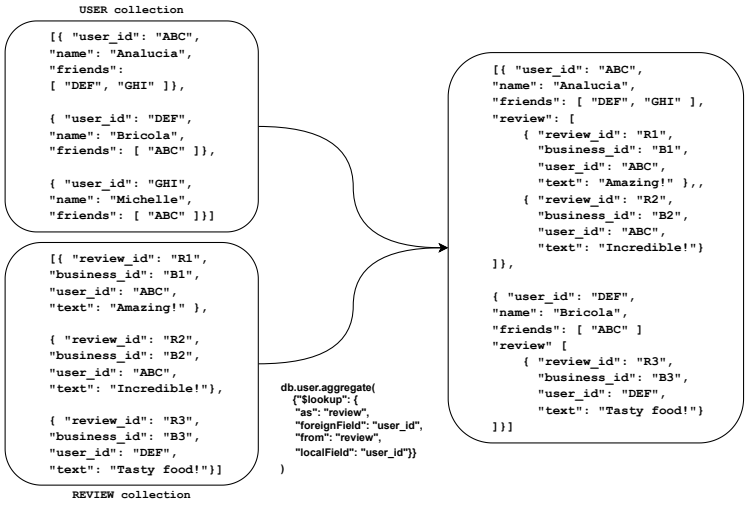


Fig. 7: $lookup stage and output example

`$unwind` Stage

In cases in which the attribute is nested inside an array, the `$match` stage can not operate directly on the attribute. To tackle this issue, we use the `$unwind` stage that "flattens" the document on the given attribute, i.e., it deconstructs an array and replicates the document for each element from the array, as seen in Figure 8. In essence, each document given as the output for this stage resembles the input document with the value of the array field replaced by an element from the array.

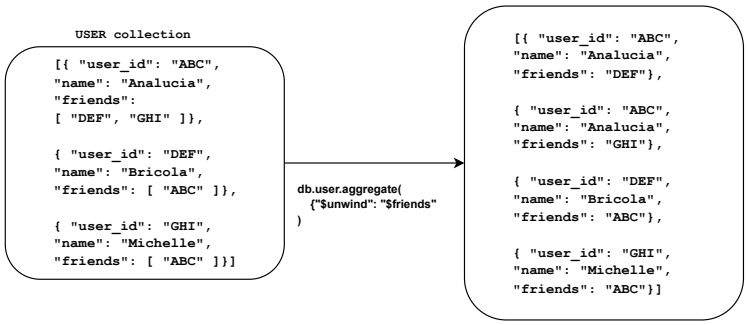


Fig. 8: $unwind query stage and output example

`$set` Stage

When executing the `$unwind` operation, MongoDB deconstructs an array and replicates the input document for each element of the array, as seen in Figure 7. To avoid rebuilding the deconstructed array, *SEREIA* duplicates the field that presents a list in its nested structure

through a `$set` stage. This operation gets the content of the attribute passed as argument and duplicates it in another attribute.

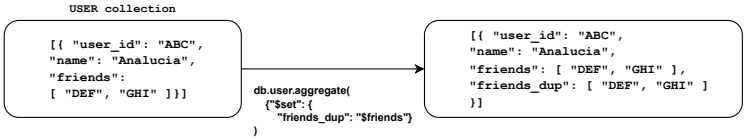


Fig. 9: $set query stage and output example

`$group` Stage

After an `$unwind` stage, it is common that multiple documents resemble the original document, as shown in Figure 8. To revert this scenario, *SEREIA* groups all the information from the original document through a `$group` stage. In this work, this stage is solely used to aid when retrieving the original documents, discarding all documents replicated due to the semantics of the `$unwind` stage. Figure 10 illustrates the output of this operator.

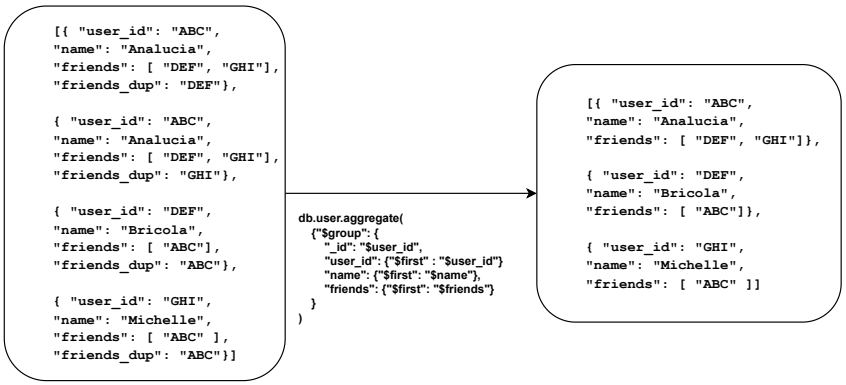


Fig. 10: $group stage and output example

7.2 Structured Query Generation Algorithm

Algorithm 1, called *MongoQueryBuilder* (MQB), is responsible for translating a CJN to a structured query. In the discussion below, we illustrate the functioning of the algorithm using the following example PLS:

$$P_1 = \texttt{USER}^V[name^{\{michelle\}}]$$
$$\downarrow$$
$$\texttt{REVIEW}^S[self^{\{reviewed\}}]$$
$$\downarrow$$
$$\texttt{BUSINESS}^V[categories^{\{italian,restaurants\}}]$$

The MQB algorithm traverses the input PLS and outputs the query that can be executed by MongoDB, along with the *base collection*, which we will define shortly. The process is straightforward and starts when MQB selects the top-most KM as the starting node and defines the previous node as `NULL` (lines 1-2). In particular, the latter is used to check whether MQB must aggregate data between collections. Taking P_1 as the input PLS, the starting node selected is $\texttt{USER}^V[name^{\{michelle\}}]$. The collection associated with this node is called the

base collection (Line 4) since it will be the first collection processed in the Aggregation Pipeline. In our running example, MQB takes `USER` as the base collection. Next, MQB defines an empty set for all visited nodes of *CJN* (Line 4). Finally, MQB calls another algorithm called *BuildStages* (Algorithm 2), which recursively traverses P and generates MongoDB stages to properly handle the collection corresponding to each node from P in the final query.

Algorithm 1: MongoQueryBuilder(P)

Input: P: A pipeline sketch
Output: MQ: A structured mongo query
$baseCollection$: base collection for aggregation

1 **let** $currNode \leftarrow$ the top-most node from P
2 **let** $prevNode \leftarrow$ `NULL`
3 **let** $C^S[A_1^{K_1^S}, \ldots, A_n^{K_n^S}]^V[A_1^{K_1^V}, \ldots, A_n^{K_n^V}]$ be the KM from $currNode$
4 $baseCollection \leftarrow C$
$visited \leftarrow \{\}$;
$MQ \leftarrow \{\}$;

5 BuildStages(MQ, $currNode$, $prevNode$, $visited$);

6 **return** MQ, $baseCollection$

The BuildStages algorithm calls several functions to build stages and adds them to the structured query being built. As described below, there is a specific function for each type of stage (Section 7.1). The algorithm's initial step consists of extracting the KM from $currNode$. Next, the algorithm checks if $prevNode$ is not `NULL` (Line 1). In this case, it generates a `$lookup` stage to indicate a join operation between the collections associated with $currNode$ and $prevNode$ (Line 2). In addition, BuildStages generates an `$unwind` stage that allows for attribute value matching in the next stage (Line 3). Since we are in the first iteration, $prevNode$ is `NULL`. In our running example, BuildStages iterates in all attributes of $currentKM$ (Line 4), i.e., $\texttt{USER}^V[name^{\{michelle\}}]$, which generates `$match` stages (Line 7). This KM only presents a single attribute *name*, and generates only one match stage as shown below.

In cases in which an attribute is located inside a list, BuildStages also generates `$set` and `$unwind` stages, only to allow value matching (Lines 5-6). In this case, after matching, BuildStages "cleans" the document by returning to its original state through a `$group` stage (Line 8). Finally, BuildStages marks the current node as visited and iterates in its adjacent nodes calling itself recursively if they were not visited (Lines 9-11).

As the adjacent node to $\texttt{USER}^V[name^{\{michelle\}}]$, BuildStages is called and passes the next node $\texttt{REVIEW}^S[self^{\{reviewed\}}]$ and $currNode$ as arguments, where the latter refers to the node previously visited. After extracting the $currentKM$, BuildStages asserts that $prevNode$ is not `NULL` and generates a `$lookup` stage to perform a join between collections. In this case, the join will be performed between the collections `USER` and `REVIEW`. After joining, BuildStages generates an `$unwind` stage to flatten the array of joined documents, in order to enable matching in the next stage, if needed. Since $currentKM$ does not present any value for matching (i.e. a SKM), this node is marked as visited and the algorithm is called for the next adjacent node. In the final iteration, BuildStages generate `$lookup` and `$unwind` stages.

Algorithm 2: BuildStages(MQ,currNode,prevNode,visited)

```
1  currentKM ← KM from currNode;
   if prevNode <> NULL then
2      MQ.append(LookupStage(prevNode, currNode));
           let C_cur be the attribute generated by $lookup in currNode
3      MQ.append(UnwindStage(C_cur));
4  foreach ⟨ attribute A_i, value K_i^V ⟩ in currentKM do
5      if hasArray(A_i) then
6          MQ.append(SetStage(A_i));
             MQ.append(UnwindStage(A_i));
7      MQ.append(MatchStage(A_i, K_i^V));
       if hasArray(A_i) then
8          MQ.append(GroupStage());
9  visited ← visited ∪ currNode;
   foreach node v ∈ getAdjacents(currNode) do
10     if v ∉ visited then
11         BuildStage(MQ, v, currNode, visited);
```

Notice that $currNode$ contains a VKM, which presents values that must be matched. For each of the attributes and values, BuildStages generates matches. Finally, after the recursive calls to BuildStages, MQB returns the Mongo Query built MQ and the base collection C, enabling *SEREIA* to issue the query against MongoDB. The MongoDB query generated for the example is given in Figure 2.

8 Experimental Evaluation

In this section, we report a set of experiments performed with *SEREIA* using datasets previously used in the literature related to document stores. The experiments aimed at submitting exploratory keyword queries to *SEREIA* to assess whether it is capable of correctly retrieving the information expected by the user, thus aiding in data discovery and exploration tasks.

8.1 Setup

We ran all experiments on a machine running Arch Linux with the following specifications: 32GB RAM, Intel® Core™ i9-10850K @ 4.9GHz. All implementations were made in Python 3.10. We used MongoDB with default configurations as document store.

Datasets

Our experiments comprised five datasets, for which we summarize the statistics in Table 2, showing the number of collections, documents, queries, and respective dataset sizes[5].

The *NBA* dataset contains data for 28 years of basketball games, ranging from 1985 to 2013. For this dataset, we generated a set of ten keyword queries. The *Twitter* dataset contains nearly 2 million documents obtained directly from the Twitter API. For this dataset, we generated

[5]Datasets available at https://github.com/bdri-ufam/sereia

eleven queries. The *DBLP* dataset contains nearly 2 million documents, containing data on conference proceedings, journal articles, PhD thesis, etc. The query set for *DBLP* was obtained from *SPARK* [22] and contains a total of ten queries. All three datasets were used in previous work for exploratory OLAP on document stores [23]. The *IMDb* dataset was obtained from Kaggle[6] and has documents containing data from approximately 45K movies released before July 2017 from the well-known Internet Movie Database (IMDb)[7]. The query set for *IMDb* has previously been used in the evaluations of keyword search system [24] and contains 26 queries. The *Yelp!* dataset was used for benchmarking a JSON parser [25] and contains rich data on businesses, as well as users and their reviews and tips from these businesses. The query set for *Yelp!* was obtained from *SQLizer* [18] and consists of 28 queries formulated in Natural Language. We adapted these queries to our experiments by removing their non-keyword terms.

Table 2: Dataset collections and query sets statistics

Dataset	No. Collections	No. Documents	Size (GB)	No. Queries
NBA	1	31.686	0.2	10
Twitter	1	1.984.049	5.1	11
DBLP	1	1.984.049	0.8	10
IMDb	2	90.939	0.2	26
Yelp!	6	12.486.440	17.4	28

Gold Standards and Evaluation Metrics

We evaluated the results of *SEREIA* in two steps: (i) quality of the query matches and pipeline sketches ranking and (ii) quality of the documents retrieved. For each step, we manually generated the relevant set of QMs, PLSs, and relevant documents to be retrieved.

To evaluate (i), we use the Precision@k ($P@k$) and Mean Reciprocal Rank. Given a keyword query Q, the value of $P_Q@K$ is 1 if the correct Query Match or Pipeline Sketches for Q appears in a position up to position K in the ranking and 0 otherwise. $P@K$ in our evaluations is the average of $P_Q@K$ for all Q in the query set. With respect to MRR, considering a keyword query Q, the value of *reciprocal ranking* for Q, RR_Q, is given by $\frac{1}{P}$, where P is the rank position of the relevant result. The MRR for a query set is the average of RR_Q for all Q in the query set. As for (ii), we measure precision, as the ratio of relevant documents among all retrieved documents, and recall, as the ratio of relevant documents retrieved among all relevant documents for a given query.

8.2 General Results

Table 3 presents statistics on the results obtained for each dataset used in the evaluation process. More specifically, considering all queries in a query set, we show the maximum (Max) and the average (Avg) number of schema-keyword matches (No. SKMs), value-keyword matches (No. VKMs), query matches (No. QMs) and pipeline sketches (No. PLSs).

Overall, the number of SKMs is low across all datasets since many queries do not present references to the datasets' schema. Naturally, the number of VKMs is higher, since most queries express keywords present in attribute values. This is the case for most queries in the *IMDb* query set that cite actors' names such as "*johnny depp*" and "*will smith*". In particular, the *DBLP* queries did not present references to the document's schema elements.

[6] https://www.kaggle.com/rounakbanik/the-movies-dataset
[7] https://www.imdb.com/

Table 3: General statistics on the results for each dataset

	No. SKMs		No. VKMs		No. QMs		No. PLSs	
Dataset	Max	Avg	Max	Avg	Max	Avg	Max	Avg
NBA	2	0.5	11	4.2	16	3.8	3	1.2
Twitter	2	0.18	48	26	199	86	29	12
DBLP	0	0	20	7.9	140	22.7	3	1.2
IMDb	4	0.5	192	34	75197	3191	5551	438
Yelp!	3	0.9	37	15	497	72	318	49

Regarding QMs, the numbers are higher due to the combinatorial process of the matching phase (Section 5). As seen in *IMDb*, the maximum number of QMs generated was 75,197. This is due to values that may be found in many different attributes and considering a single or multiple documents. For example, the keyword query "*harrison ford george lucas*" presents keywords, such as "george", that can be mapped to, for example, `credits.cast.name` or `credits.cast.character`.

Note that, in general, the number of PLSs was lower than the number of QMs in most datasets. This is explained by the pruning conditions explained in Section 6. The *Yelp!* dataset presented a different behavior, in which the maximum number of PLSs exceeds the maximum number of QMs. In this case, we see that some QMs may generate more than one PLS for a given keyword query, resulting in a larger number of PLSs. This results from the fact that *Yelp!* presents more collections that can be used when connecting KMs, increasing the number of PLSs. For instance, in the QM $\{\texttt{Business}^{S}[self^{\{businesses\}}], \texttt{User}^{V}[name^{\{bricola\}}]\}$ the keyword matches can be connected through the `Tip` or `Review` collections.

8.3 Query Match Ranking Evaluation

In this experiment, we evaluate the quality of the QM ranking algorithm based on the $P@K$ and MRR metrics. Since a single keyword query may generate several QM, *SEREIA* relies on the QM ranking algorithm to select the correcto one among those generated. In Figure 11, we show the overall evaluation of the query match ranking for each dataset.

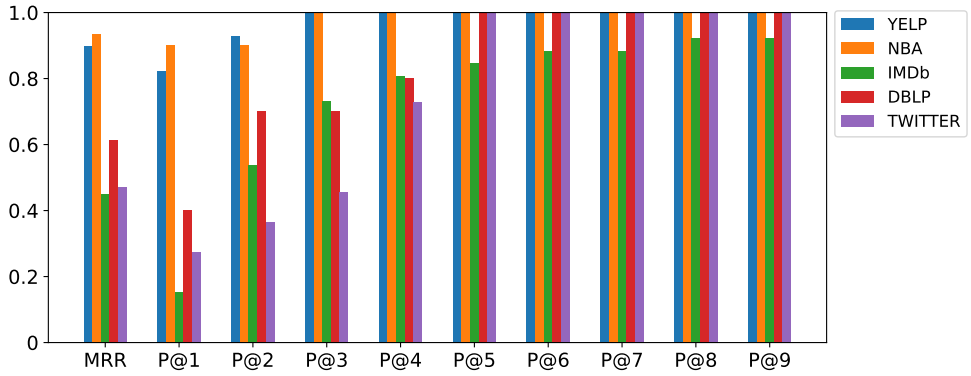


Fig. 11: MRR and $P@K$ of Query Match Ranking

In all datasets but one, *SEREIA* was able to find the correct QM at most in the 5^{th} position of the ranking. When considering the *Yelp!* and *NBA* datasets, $P@1$ is above 0.9, indicating that the correct answer is in the first ranking position for several queries. In the case of the *IMDb*,

DBLP and *Twitter* datasets, ranking the correct QM in the higher positions was harder. This is due to a handful of queries that generated several QMs since they match values scattered through many attributes. Some of these QMs, although not correct, received a high score and ranked higher than the correct one. Thus, as seen in the graph, there are cases in which the correct QM appears only in the 5^{th} position. Regarding the MRR metric, we reached a score of 0.67, which reflects the presence of almost all correct QMs within the top 3 ranking positions.

As an example of situations in which *SEREIA* was not able to rank the correct QM in the first position, consider the keyword query "*will smith*". The QM $\{\texttt{CREDITS}[\texttt{cast.name}\{will, smith\}]\}$ is the correct one for this query. However, the QM ranking algorithm assigned the top position to the QM $\texttt{MOVIES}[\texttt{overview}\{will, smith\}]$. In this situation, the term "*will*" is a verbal form frequently found in the attribute `overview`, and thus, yielded to a high score in the second QM, as detailed in Section 5.

8.4 PLS Ranking Evaluation

In this experiment, we evaluated the quality of our PLS ranking. As in the previous section, we also use the $P@k$ and MRR metrics for evaluation, shown in Figure 12. For this experiment, the generation of PLSs uses the top-N QMs based on their ranking. Considering the findings of the QM ranking experiment, we used $N = 9$.

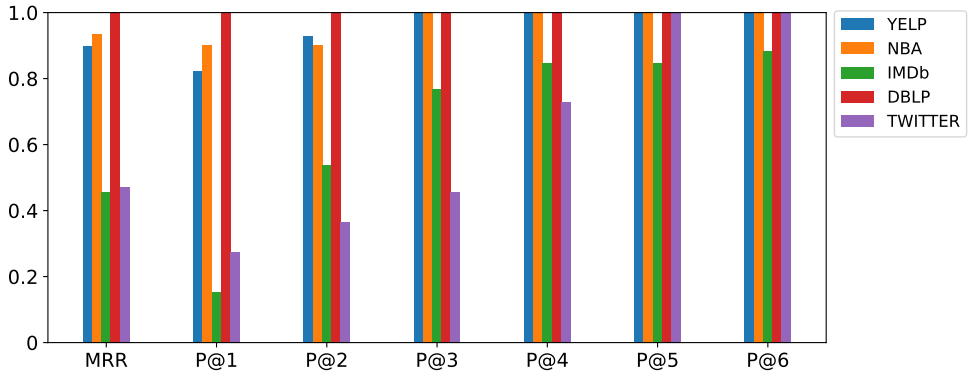


Fig. 12: MRR and $P@K$ of Pipeline Sketches

In the PLS ranking experiment, *SEREIA* ranked the correct PLS at most in the 5^{th} position of the ranking for the majority of datasets, except the *IMDb* dataset. Nonetheless, the $P@1$ metric presented better results than in the QM ranking evaluation. This improvement is more evident in the *DBLP*, where the ranking algorithm places all correct PLSs in the first position. Again, the *IMDb* dataset poses difficulties due to matching values scattered through many attributes. Furthermore, as noted earlier (Section 6.2), the PLS ranking algorithm leverages the score previously calculated by the QM ranking, leading to a scenario that is similar to the QM ranking evaluation for the *IMDb* dataset. However, both QM and PLS ranking experiments differ due to the scoring strategy employed by the PLS ranking algorithm, which favors PLSs with smaller sizes and higher scores.

8.5 Performance Evaluation

Table 4 exhibits the average execution times per phase and also the total time regarding the whole generation process. In general, the time spent in the whole process is dominated by the QM generation phase, in particular in the case of datasets with larger number of colections, i.e., *Yelp!* and *IMDB*, due to the increased number of combinations between KMs. KM generation time is low in general, and higher times are observed in *Yelp!* and *Twitter* datasets, due to matching values scattered through multiple attributes, which results in a slight increase in time. For instance, considering the keyword query "*5.0 star italian restaurants*" for the *Yelp!* dataset, the keyword "star" could be narrowed down to the number of stars of a given business, the number of stars of a given review, to a business name or a business' address. Thus, a single keyword may incur the generation of several KMs.

Table 4: Avg. Execution time for each *SEREIA* phase (ms)

Dataset	KM	QM	PLS	Total
Yelp!	11.68	160.45	9.86	182.00
NBA	2.21	2.21	0.07	4.49
DBLP	0.77	3.17	0.19	4.14
Twitter	6.29	3.00	0.29	9.60
IMDb	3.51	549.70	5.45	558.67

As for the PLS generation phase, the execution times are directly affected by the number of distinct collections referred by the generated QMs. In other words, an increased number of collections results in more possibilities when connecting KMs from a QM. This is clearly observed in the *NBA*, *DBLP* and *Twitter* datasets. Since all of them present a single collection, they require less computation and time resources. On the other hand, as the number of collections increase, so does the generation time, as in the case of the *IMDb* and *Yelp!* datasets.

Once we have the PLSs, the last step involves generating structured queries that will be executed in the underlying document store (Section 7). In Table 5 we show the average and maximum times for generation of structured queries, as well as for the execution of these queries. Overall, the maximum generation time is as low as 1 millisecond regarding the query generation, thus demonstrating the efficiency of our query generation algorithm. The overall execution times, on the other hand, are more prone to last longer due to the execution in the underlying document store. In our experiments, we saw a maximum of 9 seconds for execution time, which was in the *Yelp!* dataset. This higher time is seen in the keyword query "*businesses bricola stars 5*", in which we must join three collections (`Business`, `User` and `Review`) while also filtering documents based on attribute values, such as "*Bricola*" for the username and "*5*" as the review rating.

Table 5: Query Generation and Execution time statistics (ms)

Step	Statistics	NBA	Twitter	DBLP	IMDb	Yelp!
Generation	Avg Time	0.177	0.243	0.235	0.182	0.232
	Max Time	0.275	0.343	0.331	0.385	1.311
Execution	Avg Time	450.7	4016.0	2521.4	217.5	616.9
	Max Time	1009.0	6024.2	3669.5	483.1	9014.3

8.6 Quality of results evaluation

Once *SEREIA* generates structured queries and executes them in the underlying document store, we can assess the quality of the documents retrieved against the set of documents expected according to our gold standards and considering each dataset. In Figure 13 we evaluate this quality using the *Precision* and *Recall* metrics. In this evaluation, we consider two scenarios: (i) the Precision/Recall metrics considering the Top-1 PLS ranked by *SEREIA* and (ii) one PLS manually selected by the user from the Top-5 PLS rank. In the latter, we exhibited the top PLSs for the user and she selected the one that best suited the necessity of the keyword query. We name this approach as Assisted Ranking (AR), while the default approach we will refer as Default Ranking (DR).

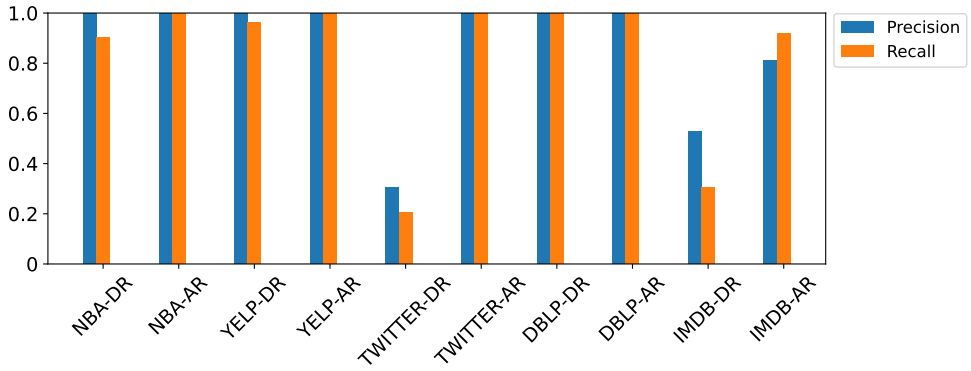


Fig. 13: Precision and Recall for *SEREIA* in each dataset

On average, the precision and recall presented by *SEREIA* using DR for the query sets were 0.76 and 0.67, respectively. Overall, *SEREIA* correctly retrieved the majority of expected documents in this scenario. In the fewer cases that affected the precision and recall metrics, the correct PLSs were not in the first rank position and, thus, *SEREIA* did not retrieve only documents expected in the query's intent. These superfluous documents derive from queries in which keywords were scattered in different document attribute values, as stated in Section 5. In contrast, adopting AR increased both metrics, achieving 0.96 and 0.98 for precision and recall, respectively. This shows that *SEREIA* was able to help the user correctly retrieve data by choosing between the top PLS ranked alternatives.

Table 6: Number of correct queries per dataset and approach

Dataset	Default Ranking	Assisted Ranking	Total
Yelp!	23 (82%)	28 (100%)	28
NBA	9 (90%)	10 (100%)	10
DBLP	10 (100%)	10 (100%)	10
Twitter	3 (27%)	11 (100%)	11
IMDb	3 (11%)	22 (85%)	26

Additionally, in Table 6, we show the number of queries correctly generated per dataset and ranking approach. We indicate for each dataset (Column 1), the number of correct queries considering the approach of the PLS ranking (Columns 2 and 3), that is, Default Ranking

and Assisted Ranking, respectively. Finally, we also indicate the total number of queries to be generated per dataset (Column 4). As noted in Figure 13, the Assisted Ranking produces better outputs since the correct query is found within the top-5 positions, as opposed to the Default Ranking approach that considers only the top-1 position.

9 Conclusions and Future Work

In this paper, we presented *SEREIA*, a system that helps users explore data in document stores via keyword queries. To our knowledge, *SEREIA* is the first system that allows exploration using keyword queries directly in a document store. Our system extends previous work on keyword search systems in relational databases [12, 13], and we proposed several changes to enable the adoption of document stores. We present a comprehensive set of experiments that we carried out using datasets and query sets previously adopted in the literature. The results of these experiments indicate that *SEREIA* is capable of helping users explore document stores, and requires only a handful of keywords as input, rather than requiring the user to generate a fully structured query in the document store query language. This is important because the time and effort spent learning how to explore data available in document stores is greater when compared to relational databases due to the semi-structured nature of the documents. By providing a more straightforward approach to exploring document stores, that is, through keyword queries, *SEREIA* facilitates data analysis in document stores and decreases the time necessary for an analyst to find relevant data.

In this work, our implementation is specific to MongoDB, a system we chose because of its popularity. However, there may be cases where developers may prefer alternative document stores, such as *CouchDB*[8] or *Couchbase*[9]. Considering its architecture, *SEREIA* can be easily adapted to other document stores. Adding support for other document stores requires two main changes: (i) implementing an adapter to retrieve data from the target document store and (ii) implementing a module to translate a CJN to the query language of the target document store.

In addition, several systems in the literature focus on handling data stored in heterogeneous data sources. Such systems, often called *polystores* [26, 27], are database management systems built on top of multiple heterogeneous storage engines, including relational databases and document stores [28]. Since with *SEREIA* we demonstrated that keywords are a viable alternative for exploring document stores, and many previous systems have also achieved the same result for relational databases, an intriguing question is whether keyword-based queries could be used as a unifying approach to allow data exploration in polystores. Another similar application scenario for such an approach is data lakes [5, 29], which presents massive stores of data, such as relational and JSON data, for future analytics. In both scenarios, leveraging data exploration through keyword search may be helpful to the user, as she would not need to know details on the organization of each distinct data source, and could then explore the contents of these sources with less difficulty. As in other tasks related to polystores and data lakes, realizing such an approach would require some form of integration between stored data items in different formats. For instance, a challenge that arises is that there is no explicit definition on how data from different sources can be joined. Techniques for finding relationships between distinct datasets have been investigated in the past literature [6, 29, 30]. In conventional approaches, techniques often leverage metrics like content and schema similarity to minimize manual labor in establishing connections among related information within data

[8] https://couchdb.apache.org/
[9] https://www.couchbase.com/

lakes. Recent advancements in Large Language Models (LLMs) offer a promising alternative. Researchers are increasingly utilizing LLMs to automatically analyze documents stored in data lakes. These models not only interpret the content but also facilitate the generation of links between related data [31]. Furthermore, LLMs can automate the process of code generation for data extraction tasks [32]. By integrating techniques like these into our framework, we can explore its potential usefulness in scenarios that involve polystores or data lakes.

Acknowledgements

This work was supported by the Foundation for Research Support of the State of Amazonas (FAPEAM) - POSGRAD 2021.

References

[1] Sadalage, Fowler: Nosql distilled: A brief guide to the emerging world of polyglot persistence (2012) **13**, 978–0321826626 (2012)

[2] DiScala, Abadi: Automatic generation of normalized relational schemas from nested key-value data. In: Proc. of the 2016 Intl. Conf. on Management of Data (2016)

[3] Tahara, *et al.*: Sinew: a sql system for multi-structured data. In: Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data, pp. 815–826 (2014)

[4] Chasseur, *et al.*: Enabling json document stores in relational systems. In: Proc. of the 16th Intl. Workshop on the Web and Databases, vol. 13, pp. 1–6 (2013)

[5] Terrizzano, I.G., Schwarz, P.M., Roth, M., Colino, J.E.: Data wrangling: The challenging yourney from the wild to the lake. In: CIDR (2015)

[6] Fernandez, R.C., Abedjan, Z., Koko, F., Yuan, G., Madden, S., Stonebraker, M.: Aurum: A data discovery system. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1001–1012 (2018). IEEE

[7] Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., *et al.*: Presto: Sql on everything. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1802–1813 (2019). IEEE

[8] Hamadou, H.B., Ghozzi, F., Péninou, A., Teste, O.: Schema-independent querying for heterogeneous collections in nosql document stores. Information Systems **85**, 48–67 (2019)

[9] Liu, *et al.*: Closing the functional and performance gap between sql and nosql. In: Proc. of the 2016 Intl. Conf. on Management of Data, pp. 227–238 (2016)

[10] Rezig, E.K., Bhandari, A., Fariha, A., Price, B., Vanterpool, A., Gadepally, V., Stonebraker, M.: Dice: Data discovery by example. Proceedings of the VLDB Endowment **14**(12), 2819–2822 (2021)

[11] Helal, A., Helali, M., Ammar, K., Mansour, E.: A demonstration of kglac: a data discovery and enrichment platform for data science. Proceedings of the VLDB Endowment **14**(12),

2675–2678 (2021)

[12] Oliveira, P., Silva, A., Moura, E.: Ranking candidate networks of relations to improve keyword search over relational databases. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 399–410 (2015). IEEE

[13] Oliveira, P., Silva, A., Moura, E., Rodrigues, R.: Match-based candidate network generation for keyword queries over relational databases. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1344–1347 (2018). IEEE

[14] Oliveira, P.S., Da Silva, A., Moura, E., De Freitas, R.: Efficient match-based candidate network generation for keyword queries over relational databases. IEEE Transactions on Knowledge and Data Engineering (2020)

[15] Hristidis, Papakonstantinou: Discover: Keyword search in relational databases. In: VLDB'02: Proc. of the 28th Intl. Conf. on Very Large Databases, pp. 670–681 (2002)

[16] Martins, P., Silva, A.S., Afonso, A., Cavalcanti, J., Moura, E.: Supporting schema references in keyword queries over relational databases. IEEE Access **11**, 92365–92390 (2023) https://doi.org/10.1109/ACCESS.2023.3308908

[17] Li, F., Jagadish, H.V.: Nalir: an interactive natural language interface for querying relational databases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 709–712 (2014)

[18] Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: Sqlizer: Query synthesis from natural language. Proc. ACM Program. Lang. **1**(OOPSLA) (2017) https://doi.org/10.1145/3133887

[19] Hu, X., Duan, J., Dang, D.: Natural language question answering over knowledge graph: the marriage of sparql query and keyword search. Knowledge and Information Systems **63**, 819–844 (2021)

[20] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval: The Concepts and Technology Behind Search, 2nd edn. Addison-Wesley Publishing Company, USA (2008)

[21] Mesquita, F., Silva, A.S., Moura, E.S., Calado, P., Laender, A.H.: Labrador: Efficiently publishing relational databases on the web by using keyword-based query interfaces. Information Processing & Management **43**(4), 983–1004 (2007)

[22] Luo, Y., Wang, W., Lin, X.: Spark: A keyword search engine on relational databases. In: 2008 IEEE 24th International Conference on Data Engineering, pp. 1552–1555 (2008). https://doi.org/10.1109/ICDE.2008.4497619

[23] Chouder, M.L., Rizzi, S., Chalal, R.: Exodus: Exploratory olap over document stores. Information Systems **79**, 44–57 (2019) https://doi.org/10.1016/j.is.2017.11.004 . Special issue on DOLAP 2017: Design, Optimization, Languages and Analytical Processing of

[24] Coffman, J., Weaver, A.C.: A framework for evaluating database keyword search strategies. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, pp. 729–738 (2010). ACM

[25] Li, Y., Katsipoulakis, N.R., Chandramouli, B., Goldstein, J., Kossmann, D.: Mison: A fast json parser for data analytics **10**(10), 1118–1129 (2017) https://doi.org/10.14778/3115404.3115416

[26] Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The bigdawg polystore system. ACM Sigmod Record **44**(2), 11–16 (2015)

[27] Deng, D., Fernandez, R.C., Abedjan, Z., Wang, S., Stonebraker, M., Elmagarmid, A.K., Ilyas, I.F., Madden, S., Ouzzani, M., Tang, N.: The data civilizer system. In: Cidr (2017)

[28] Alotaibi, R., Cautis, B., Deutsch, A., Latrache, M., Manolescu, I., Yang, Y.: Estocada: towards scalable polystore systems. Proceedings of the VLDB Endowment **13**(12), 2949–2952 (2020)

[29] Ouellette, P., Sciortino, A., Nargesian, F., Bashardoost, B.G., Zhu, E., Pu, K.Q., Miller, R.J.: Ronin: data lake exploration. Proceedings of the VLDB Endowment **14**(12) (2021)

[30] Bogatu, A., Fernandes, A.A., Paton, N.W., Konstantinou, N.: Dataset discovery in data lakes. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 709–720 (2020)

[31] Chen, Z., Gu, Z., Cao, L., Fan, J., Madden, S., Tang, N.: Symphony: Towards natural language query answering over multi-modal data lakes. In: Conference on Innovative Data Systems Research, CIDR, pp. 8–151 (2023)

[32] Arora, S., Yang, B., Eyuboglu, S., Narayan, A., Hojel, A., Trummer, I., Ré, C.: Language models enable simple systems for generating structured views of heterogeneous data lakes. arXiv preprint arXiv:2304.09433 (2023)