Similarity-Slim Extension: Reducing Financial and Computational Costs of Similarity Queries in Document Collections in NoSQL Databases

William Zaniboni Silva[a], Igor Alberte Rodrigues Eleutério[b], Larissa Roberta Teixeira[c], Agma Juci Machado Traina[d] and Caetano Traina Júnior[e]

Institute of Mathematics and Computer Sciences (ICMC), University of São Paulo, São Carlos, Brazil

Keywords: Similarity Query, NoSQL, Metric Access Methods, Cloud-Based Storage, Billing Reduction.

Abstract: Several popular cloud NoSQL data stores, such as MongoDB and Firestore, organize data as document collections. However, they provide few resources for querying complex data by similarity. The comparison conditions provided to express queries over documents are based only on identity, containment, or order relationships. Thus, reading through an entire collection is often the only way to execute a similarity query. This can be both computationally and financially expensive, because data storage licenses charge for the number of document reads and writes. This paper presents Similarity-Slim, an innovative extension for NoSQL databases, designed to reduce the financial and computational costs associated with similarity queries. The extension was evaluated on the Firestore repository as a case study, considering three application scenarios: geospatial, image recommendation and medical support systems. Experiments have shown that it can reduce costs by up to 2,800 times and speed up queries by up to 85 times.

1 INTRODUCTION

Several popular cloud NoSQL data stores, such as MongoDB (MongoDB, 2023) and Firestore (Kesavan et al., 2023; Google, 2023a), organize data as document collections. The query costs are associated with the number of read and write operations performed on the documents: for example, reading 100,000 documents in Firestore costs US$ 0.06, as shown in Table 1.

Table 1: Firestore costs to handle documents in USA(Google, 2023h).

Operation over 100,000 documents	cost
Read	US$ 0.06
Write	US$ 0.18
Delete	US$ 0.02

Although NoSQL stores provide powerful resources to retrieve data based on relationships of identity, order, containment, and even some support for spatially located queries (Koutroumanis and Doulkeridis, 2021), include indexing structures to accelerate them (Qader et al., 2018), few resources, if any, are provided to query by similarity complex data such as images, geolocated objects, and texts.

Often, reading the entire collection of documents is the only way to perform similarity queries. Considering that the licenses charge for the number of document operations, this can turn to be expensive. To the best of the authors' knowledge, there is no work in the literature focused on optimizing the amount of document reads/writes and the associated financial cost to perform similarity queries in NoSQL document stores.

This work aims at creating an extension for NoSQL data stores, called Similarity-Slim, which reduces the amount of reads when performing similarity queries on large document collections. As a case study, we also perform experiments to evaluate the extension using the Firestore data store. The similarity comparisons are evaluated using a distance function defined by the application. The search algorithm retrieves exact answers, meaning that when the k nearest neighbors are requested, the response is the correct, not an approximate answer.

The experiments were conducted on three real-world datasets: Geonames (Unxos GmbH, 2023),

[a] https://orcid.org/0000-0003-2961-9627
[b] https://orcid.org/0009-0007-3987-8880
[c] https://orcid.org/0009-0007-9917-4404
[d] https://orcid.org/0000-0003-4929-7258
[e] https://orcid.org/0000-0002-6625-6047

Silva, W., Eleutério, I., Teixeira, L., Traina, A. and Traina Júnior, C.
Similarity-Slim Extension: Reducing Financial and Computational Costs of Similarity Queries in Document Collections in NoSQL Databases.
DOI: 10.5220/0012606300003690

In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS 2024) - Volume 1*, pages 95-106
ISBN: 978-989-758-692-7; ISSN: 2184-4992

DeepLesion (Yan et al., 2018; Yan et al., 2019) and FeatSet+ (Cazzolato et al., 2022). They contain data from a variety of complex domains (geospatial points, sets, and images) of varying cardinalities and dimensionalities. The performance of the queries using our solution was compared with equivalent queries executed by a sequential scan (i.e., reading the entire collection). Briefly, the main contributions of this work are as follows:

- The new Similarity-Slim extension, which is employed to optimize the time and reduce the cost of similarity queries in cloud-based NoSQL document stores.
- The analysis of a variety of case studies to validate the extension in geospatial application, image recommendation, and medical support systems.
- Analysis of a case study that validates the use of the extension on Google Firestore.

The remainder of the paper is structured as follows: Section 2 presents concepts required to understand this work; Section 3 describes the proposed extension; Section 4 illustrates the experiments, and Section 5 presents our conclusions and future work.

2 BACKGROUND AND RELATED WORK

This section shows the basic concepts required to understand this paper. Section 2.1 presents the definition of similarity queries, Section 2.2 illustrates the concepts that allow optimizing them, Section 2.3 presents a brief introduction to the Firestore infrastructure, and Section 2.4 reviews relevant related works.

2.1 Similarity Queries

Similarity queries perform comparisons based on the similarity between pairs of elements, which can be evaluated, for example, by a **distance function** d that measures the similarity as a real number that is smaller for more similar pairs. In this work, we call "complex" the data that, to be compared, requires the definition of how to measure similarity – in fact, at least one distance function, as there are usually several ways to assert similarity even among the same objects.

Many distance functions are defined in the literature (Deza et al., 2009; Wilson and Martinez, 1997), for different data domains, such as: the Manhattan distance to evaluate similarity among dimensional arrays (such as the features extracted from images) (Zhang and Lu, 2003); the Jaccard distance for sets (e.g. sets of keywords) (Niwattanakul et al., 2013) and the Orthodromic distance for geospatial points (Cong and Jensen, 2016). Figure 1 visually shows those distance functions applied to two complex elements A and B.

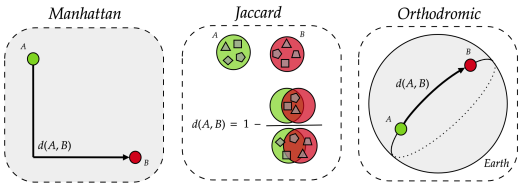

Figure 1: Some common distance functions.

A similarity query is defined by specifying a query center s_q, a similarity comparison operator (Barioni et al., 2011) and a threshold. There are two basic operators: the **Similarity Range** (*Rg*), whose threshold is a similarity radius ξ; and the *k*-**Nearest Neighbors** (*kNN*), whose threshold is the amount of elements k. A **Range Query** retrieves the elements whose similarity to s_q does not exceed ξ. A *k*-**Nearest Neighbors query** retrieves the k elements nearest to s_q.

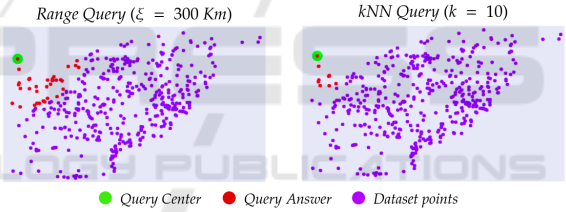

Figure 2: Similarity queries in a geospatial application.

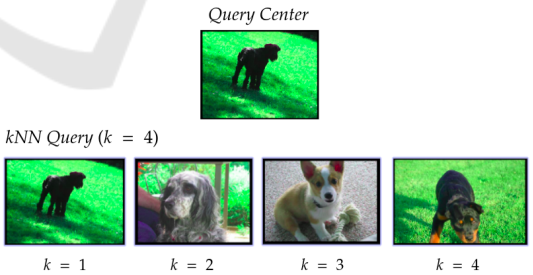

Figure 3: Similarity query in an image recommendation system.

Figures 2 and 3 exemplify similarity queries in a geospatial application and in an image recommendation system, respectively. Figure 2 shows a subset of the Geonames dataset (Unxos GmbH, 2023) queried by a range (left) and a *kNN* query (right) using the Orthodromic distance to measure similarity between geolocated points. In Figure 3, the similarity between images of dogs (Cazzolato et al., 2022) is measured

using the Manhattan distance to execute a *kNN* query with $k = 4$.

2.2 Metric Access Methods

A distance function d over a data domain $\mathcal{M}$ is called a metric when it satisfies the following properties for any elements $a, b, c \in \mathcal{M}$.

- Non-negativity: $d(a,b) \geq 0$.
- Identity of Indiscernibles: $d(a,b) = 0$ iff a and b are the same element.
- Symmetry: $d(a,b) = d(b,a)$.
- Triangular inequality: $d(a,b) \leq d(a,c) + d(c,b)$.

When these conditions are met, $\mathcal{M}$ is said to be a metric space under d. Those properties are useful to create indexing structures, called **Metric Access Methods (MAM)**, which can greatly speed up similarity queries. There are many MAMs described in the literature (Shimomura et al., 2021; Chen et al., 2022), such as the M-Tree (Ciaccia et al., 1997) and the Slim-Tree (Traina-Jr et al., 2000). They store data in fixed-size memory pages (or store a maximum number c of elements per page) in a hierarchical structure, partition the metric space into metric balls, and support dynamic updates. The Slim-Tree is an evolution of the M-Tree, which seeks to reduce the overlap of the subspaces covered by nodes in the same hierarchical level (Traina-Jr et al., 2000).

Similarity queries are executed on a Slim-Tree using the following algorithms:

- Range queries are executed with a branch-and-bound algorithm. It descends from the root to the leaves using the threshold ξ and the triangular inequality property to evaluate whether each subtree can be pruned by ensuring that its covered subspace does not overlap the query ball.
- The *kNN* query is performed by the best-first algorithm (Roussopoulos et al., 1995): the nodes are visited following a single priority queue that searches in the sub-trees for the elements closest to the query center s_q. A dynamic threshold ξ assumes the distance value from s_q to the k_{th} element already found. The threshold makes it possible to prune sub-trees using the triangular inequality.

2.3 Firestore Infrastructure

Firestore is a NoSQL document store made available by Google for mobile and web application development (Google, 2023a). It stores data as key/value pairs within documents, organizing documents into collections. Firestore does not impose a schema on the documents, making them highly customizable. It supports a wide range of data types, including boolean, bytes, date and time, floating-point numbers, geographical points, integers, arrays, maps, null values, and text strings. Each document in a collection is assigned a unique ID, and each document can store up to 1 megabyte of data.

An application can either retrieve all documents in a Firestore collection or selectively fetch only those that meet specific criteria. In the latter approach, the queries must include conditions based on the key-value pairs within the documents. It's worth noting that Firestore queries utilize indexes that are automatically generated for all keys when a new document is added.

Cloud Functions (Google, 2023b) are employed to deploy backend code that manipulates data and responds with the corresponding updates in Firestore. This also includes either reading and processing each new document added to a collection or reading the entire collection. It is also possible to combine the resources of Cloud Functions and Firestore to create extensions to publish and use new features in the data store (Google, 2023f). Usually, these extensions are used to connect third-party resources to the data store: for example, they can provide full-text search (Google, 2023d), semantic search (Google, 2023e), and approximate matches in vector similarity search (Google, 2023c).

Reading/writing documents in Firestore incurs costs, as shown in Table 1 and detailed in the Firestore pricing documentation (Google, 2023h).

2.4 Optimizing Similarity Queries on Data Stores

In the literature, the main focus on optimizing similarity queries in data stores aims at reducing query time. Works like MSQL (Lu et al., 2017), SIREN (Barioni et al., 2006), and RAFIKI (Nesso et al., 2018) use indexing structures to speed up similarity queries in a Database Management System (DBMS). For example, MSQL organizes the complex data using a B^+-Tree and the others using a Slim-Tree.

In the NoSQL domain, there is a great focus on how to perform similarity queries over big data. For example SigTrac (Damaiyanti et al., 2017) targets similarity queries over road traffic data using MongoDB, (Kim et al., 2018) and (Kim et al., 2020) studies how to support the whole lifecycle of a similarity query in Apache AsterixDB (The Apache Software foundation, 2023). TrajMesa (Li et al., 2020) focuses on queries over trajectory data domains, and (Karras

Table 2: Main differences between Similarity-Slim and related works.

Method	NoSQL document collection domain	Applied over any metric domain data	Exact Range and kNN queries	Focused on billing reduction
MSQL (Lu et al., 2017)	No	Yes	No	No
SIREN (Barioni et al., 2006) RAFIKI (Nesso et al., 2018)	No	Yes	Yes	No
SigTrac (Damaiyanti et al., 2017)	Yes	No	No	No
(Kim et al., 2018) (Kim et al., 2020)	No	No	No	No
TrajMesa (Li et al., 2020)	No	No	Yes	No
(Karras et al., 2022)	No	No	Yes	No
Similarity-Slim	Yes	Yes	Yes	Yes

et al., 2022) works with similarity queries in spatial data. Developing resources to help execute similarity queries over spatial data in NoSQL stores are presented and described in (Gonçalves et al., 2021) and (Coşkun et al., 2019).

Table 2 summarizes the main differences between the related works above and our solution based on four criteria: whether it is used in document collection NoSQL store; applied over any generic metric domain data; exact similarity queries; and focus on billing reduction. To the best of the authors' knowledge, until now, there is no work focused on optimizing the billing of similarity queries in cloud document stores. This work aims at closing this gap, presenting the Similarity-Slim Extension, which transfers a battle-tested technology to a new problem domain: transfer a MAM initially developed for relational databases to a cloud-based NoSQL document store, aiming at reducing the financial query cost of similarity queries.

3 THE PROPOSED EXTENSION

In this paper, we introduce an innovative extension, called the Similarity-Slim, designed to significantly reduce the financial costs associated with executing similarity queries over a data collection T_S stored in a NoSQL Store, such as the Google Cloud Firestore. We assume that the queries involve comparisons based on a complex attribute S, which is a component of every individual document $Doc_i \in \mathsf{T}_S$.

The size limit of a document in Firestore is significantly larger than the size required to store each document Doc_i. Thus, the basic idea for reducing the number of read operations is to concatenate multiple Doc_i into a single *c*oncatenated Firestore document Doc_c. However, for this to be effective, the documents stored together must be ones that will also need to be read together during the queries - a random concatenation will require reading many scattered Doc_c documents, making the process even worse.

The central idea of our extension is to integrate a MAM into the NoSQL store and to use its structure to identify the objects to be stored together, consolidating the complex attributes S from multiple individual documents Doc_i into a single composite document Doc_c: we define c as the maximum number of complex attribute values that are consolidated together in the same document Doc_c. Provided the concatenated values of the complex attributes from the Doc_c documents are meaningful to answer a query, multiple read operations on the data collection T_S can be transformed into fewer read operations on a new collection T_P that store the documents Doc_c.

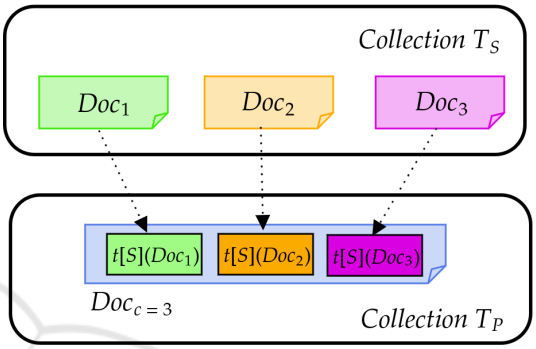

Figure 4: Concatenating $c = 3$ document's complex attributes from collection T_S in another document on collection T_P.

Figure 4 shows an example of the main idea of this extension: instead of reading all three documents from collection T_S, it is necessary only one read on collection T_P to obtain all complex attributes $S = t[S](Doc_i)$, $i \in [1,3]$.

Our extension uses the Slim-tree (Traina-Jr et al., 2000) MAM to select the documents that have the complex attributes that are worth storing together. The main reasons to choose it are:

- When deployed within an RDBMS, a Slim-tree answers similarity queries requiring significantly fewer accesses to external memory, often being the best option regarding this property (Traina-Jr et al., 2000). So, aiming at the objective of this work, the Slim-Tree can reduce the number of ac-

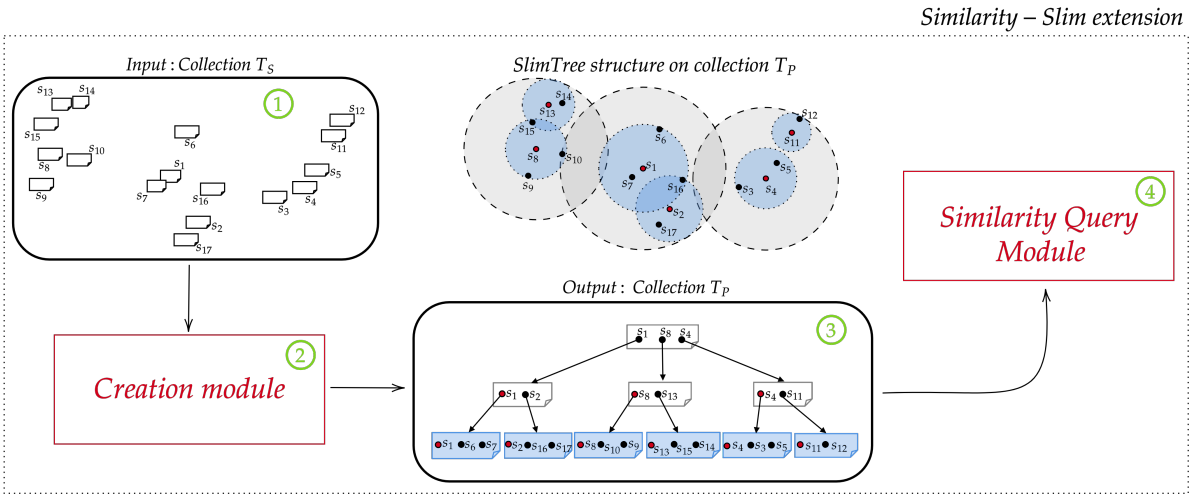

Figure 5: Similarity-Slim framework applied over a collection T_S.

cesses (reads) to documents in the NoSQL data stores domain competitively with or better than the other compared MAMs.

- Its structure uses two types of nodes (index and leaf nodes) well-tailored for storage in a document store. Correspondingly, each document Doc_c stored in T_P will always be either an index Doc_{ci} or a leaf Doc_{cl} document.
- It allows customizing the maximum number c of documents that are worth storing together in a node.
(Notice that the effective number of elements in each node can be smaller than c).

Every original document identifier Id_{sc} and S value from document $Doc_i \in T_S$ is stored in a leaf document in T_P, whereas the index documents store only copies of S values existing in a few documents from T_S: just those required to create the structure.

In short, the extension is responsible for concatenating and indexing the documents from the collection T_S in another collection T_P using the complex values S from each $Doc_i \in T_S$. Therefore, instead of performing the similarity query reading every document in collection T_S, it is performed first navigating in T_P and only reading the Doc_i documents required to be in the answer. Figure 5 shows a sketch of how Similarity-Slim works in four steps.

- Step 1: Shows the input collection T_S using a dataset with 17 documents $Doc_i, i \in [1,17]$. The value of the complex attribute $t[S]$ in each document is shown as $s_i = t[S](Doc_i)$ in a two-dimensional representation.
- Step 2: Consist of the module responsible by indexing, concatenating the complex attributes $t[S]$, and generating the documents from collection T_P.
- Step 3: Shows the output collection T_P. As can be seen, the documents from T_S are concatenated into 6 documents (in blue), while the other 4 documents (in gray) are used to index them.
- Step 4: Consist of the module responsible for performing the optimized similarity queries on collection T_P.

Similarity-Slim comprises two main modules: a create module and a similarity query module, which are described following.

3.1 The Create Module

The create module is responsible for indexing the T_S collection, creating the T_P collection. The application must define how to measure the similarity between documents $Doc_i \in T_S$ using the value $t[S](Doc_i)$ of attribute S in each document. The indexing process uses the Slim-Tree creation algorithm to structure the data in T_S into T_P. When the index collection T_P does not yet exist, a new one is built from scratch, and the complete collection T_S is loaded. Otherwise, each new document is added to both T_S and T_P collections, i.e., T_P not need to be rebuilt, just updated with the S value and document identifier Id_{sc} from the new document. The financial cost associated with this module comes from reading documents from T_S and performing read and write operations on T_P.

Figure 5 shows a two-dimensional representation of 17 documents in an Euclidean space and a hierarchical model of them in a Slim-Tree using a maximum number $c = 3$ of elements per document in T_P: leaf documents (Doc_{cl}) are displayed in blue and index documents (Doc_{ci}) are displayed in light gray. Every document identifier from collection T_S is stored in a

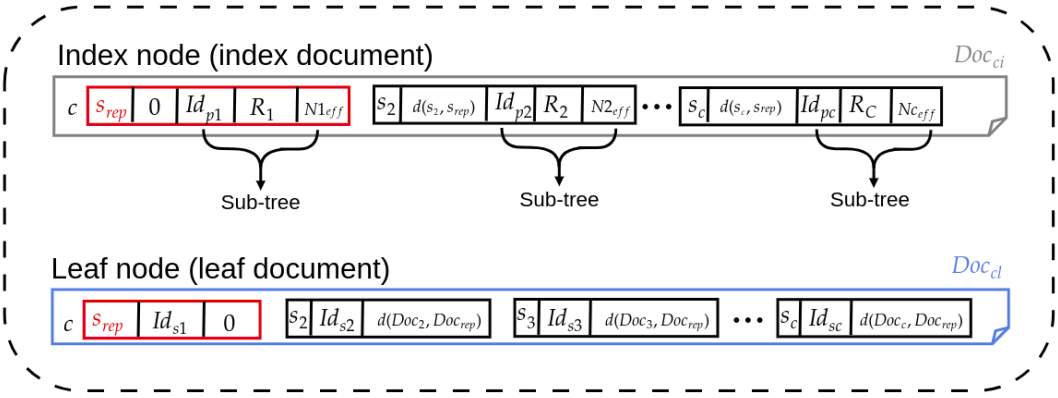

Figure 6: Slim nodes (documents) on collection T_p.

leaf document in T_P.

Figure 6 shows how the complex data are kept in each type of document in T_P. There is a representative complex value s_{rep} from attribute S, defined by the Slim-Tree's creation algorithm for creating the indexing structure, for both types (displayed as a red dot in Figure 5). Each index document $Doc_{ci} \in \mathsf{T}_P$ also stores the distances $d(s_i, s_{rep})$ from the value $t[s](Doc_i)$ of each element that it stores to its representative s_{rep} and each leaf document $Doc_{cl} \in \mathsf{T}_P$ stores the corresponding distances $d(Doc_i, Doc_{rep})$. The information about each sub-tree is stored in the corresponding index documents, including: the sub-tree covering radius R_c, the number of elements in the sub-tree Nc_{eff}, and the document identifier Id_{pc} of the sub-tree root in collection T_P. The leaf documents include the document identifiers Id_{sc} of the corresponding documents in collection T_S.

3.2 The Similarity Query Module

This module is responsible for executing the similarities queries. Every similarity search intended to be executed over collection T_S can now be executed over collection T_P using the algorithms introduced in Section 2.2, and detailed below on Sections 3.2.1 and 3.2.2.

Each query is posed by specifying the following parameters: the query center s_q and either the search radius ξ for a range query, or the amount k for a *kNN* Query. The query module retrieves the list of documents identifier Id_{sc} that satisfies the similarity condition. Now, the query answer can be returned by reading those documents from collection T_S. Therefore, the financial cost of this module is associated with reading the documents from the Slim-Tree structure in collection T_P and, if there are documents as query answers, reading them in collection T_S.

3.2.1 Range Query

Algorithm 1 shows how the range query is performed over the collection T_P to obtain the list *result* of document identifiers Id_{sc} that satisfies the similarity condition. It receives as input the document identifier Id_{pc} from the document root of the Slim-Tree on collection T_P and the query ball $Q =< s_q, \xi >$ (*result* is pre-initialized as empty). The document doc_c with identifier Id_{pc} is read and the algorithm evaluates each element s_i stored in the document; if doc_c is an index and the sub-tree centered at s_i cannot be pruned, the algorithm is called recursively passing the document identifier Id_{pi} as the document root of the respective sub-tree (lines 5-7); if it is a leaf and s_i is covered by the query ball Q, then the document identifier Id_{si} is added to the *result* list (lines 11-13). After that, all documents from *result* are read on collection T_S.

Algorithm 1: Range query over collection T_P.

```
1:  procedure RANGEQUERY(Id_pc, Q, result)
2:      doc_c ← read document with identifier Id_pc
3:      if doc_c is an index document then
4:          for each s_i in doc_c do
5:              if sub-tree centered at s_i cannot be pruned then
6:                  return rangeQuery(Id_pi, Q, result)
7:              end if
8:          end for
9:      else                                   ▷ doc_c is a leaf document
10:         for each s_i in doc_c do
11:             if s_i is covered by the query ball Q then
12:                 add Id_si in result
13:             end if
14:         end for
15:     end if
16:     return result
17: end procedure
```

Figure 7 shows an example of a Range Query performed on the T_S collection: three documents from the T_P collection and two documents from the T_S collection must be read to obtain all documents that sat-

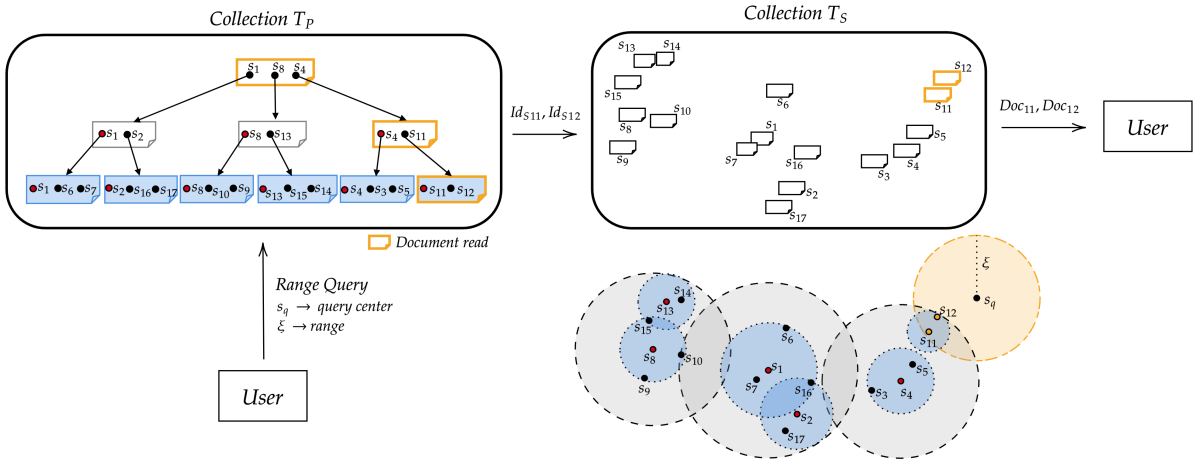

Figure 7: Similarity query module - example of a Range Query performed over collection T_S.

isfy the query parameters.

3.2.2 *kNN* Query

Algorithm 2 illustrates how the *kNN* query is performed over collection T_P. It receives as input the document identifier Id_{pc} from the document root of the Slim-Tree on collection T_P, the query center s_q, and the amount k of documents to be returned. It starts initializing the dynamic query range (d_k) as ∞ and the priority list (PQ) as empty (line 3). At each iteration, a document of the structure is read and each of its elements s_i is analyzed. If it is an index document and the ball centered at s_i cannot be pruned, then the document identifier Id_{pi} from the root document of the corresponding sub-tree is added to the priority queue using the distance between s_i and the query center (lines 7-9) as its priority. If it is a leaf document and s_i is covered by the query ball, then (s_i, Id_{si}) is added to the *result* list (line 14). If *result* has more than k elements, the farthest one is removed and the *radius* of the query ball is updated to the value of the distance between the query center and the element at position k (lines 18-20). The choice of the document that is analyzed at each iteration of the algorithm is done by the single priority queue, so it selects the document with the lowest priority that contains an intersection between its elements with the query ball (line 25-27). After that, all k documents from *result* are read on collection T_S.

Algorithm 2: *kNN* query over collection T_P.

```
1: procedure KNNQUERY(Id_pc, s_q, k)
2:     doc_c ← read document with identifier Id_pc
3:     d_k ← ∞, PQ ← empty, result ← empty
4:     repeat
5:         if doc_c is an index document then
6:             for each s_i in doc_c do
7:                 if sub-tree centered at s_i cannot be pruned then
8:                     add (s_i, Id_pi) into PQ with priority d(s_q, s_i)
9:                 end if
10:            end for
11:        else                                   ▷ doc_c is a leaf document
12:            for each s_i in doc_c do
13:                if s_i is covered by the query ball < s_q, d_k > then
14:                    add (s_i, Id_si) into result
15:                    if |result| > k then
16:                        remove the element k+1 from result
17:                    end if
18:                    if |result| = k then
19:                        d_k ← d(s_q, result[k](s_i))
20:                    end if
21:                end if
22:            end for
23:        end if
24:        repeat
25:            Id_pc ← PQ[0](Id_pi)
26:        until intersection of PQ[0](s_i) with the query ball < s_q, d_k > is
       not null or PQ is empty
27:        doc_c ← read document with identifier Id_pc
28:    until PQ is empty
29:    return result
30: end procedure
```

4 EXPERIMENTS

We evaluated the Similarity-Slim extension for three different applications. They employ datasets with varying cardinalities (n), dimensionality (E), and distance functions (d), covering many meaningful use cases. Each dataset is stored as a collection with one document in Firestore per original document.

- **Geo-Spatial Application.** We use a subset of $n = 1,000,000$ elements from the Geonames dataset (Unxos GmbH, 2023). It contains geospatial points and information about the corresponding

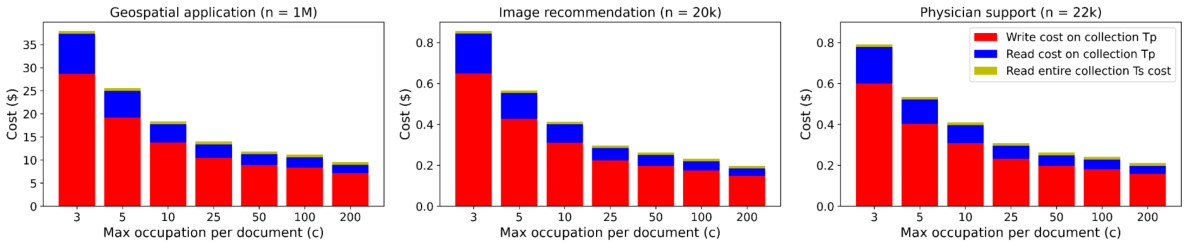

Figure 8: Financial cost of creating Slim-Tree varying the number of elements per node (c).

locations. The complex data has two dimensions (latitude and longitude), and we assume that the similarity is their geographic distance measured by the Orthodromic distance function.

- **Image Recommendation.** We use a dataset of features extracted from $n = 20,580$ images of dogs (Cazzolato et al., 2022). The complex data is the color layout characteristics extracted, which has 16 dimensions, and the similarity is measured using the Manhattan distance function.
- **Physician Diagnosis Support System.** We use the DeepLesion dataset (Yan et al., 2018; Yan et al., 2019). It contains sets of tags from annotated lesions identified on CT images and other information about patients. There are $n = 22,450$ elements in the dataset. The complex data are adimensional sets of tags whose similarity is measured using the Jaccard distance function.

Notice that Firestore must now store two document collections: T_S with the original, complete documents including the complex data, and T_P with the indexing structure and only the complex data from the original documents.

The experiments evaluate useful metrics for similarity queries: the query time, the number of similarity calculations and, most importantly for this extension, the financial costs associated with the creation and similarity query modules.

The experiments varied the maximum number of elements per Slim-Tree node (c), and computed the financial costs to handle the documents, as shown in Table 1, corresponding to the Firestore costs in EUA (multi-region) (Google, 2023h).

The Similarity-Slim extension was implemented in Python 3.11 and is available as open-source software in our GitHub (William Zaniboni Silva, 2024) ready to be deployed as a Firestore backend using Cloud Functions. The experiments were made using a Google-provided Firestore Emulator (Google, 2023g) running on a Dell-G3 computer with an Intel Core i7-8750H 2.20 GHzx12 processor, 16GB of RAM and 480 GB SSD, under the Ubuntu 20.04.4 LTS operating system.

Section 4.1 shows experiments performed on the create module, and Section 4.2 presents the results obtained evaluating the similarity query module. The metrics used to evaluate the creation module are the average of 10 index creation operations, shuffling the document ordering in the dataset. The metrics obtained from the query module correspond to the average of 20 distinct queries performed with random centers. The averages of the financial cost in the queries were scaled for 10,000 queries for better visualization.

4.1 The Create Module Costs

The first experiments evaluated the cost of creating a Slim-Tree structure. As explained in Section 3.1, this cost is related to reading the entire T_S collection and reading/writing the Slim-Tree documents in the T_P collection.

Figure 8 shows the financial cost to create a Slim-Tree for each application. As can be seen, the financial cost decreases with increasing values of c. For example, in the geospatial application, for $c = 3$, creating the Slim-Tree costs around $38, and for $c = 100$, the cost drops to $11. As it can be seen in red, the total cost is steadily dominated by the cost to write on the documents from collection T_P: it happens because the Slim-Tree's create algorithm usually performs more write than read operations and the financial cost of a write is 3 times more expensive than a read. For instance, Figure 9 shows the ratio between the number of writes and reads that are performed on the creation of collection T_P: the ratio increases with increasing c and almost stabilizes at 1.3 for $c = 200$ in every application.

Figure 10 shows the number of documents that were created for the Slim-Tree structure, i.e., the number of documents on the collection T_P that concatenates and indexes the documents from T_S. For example, in the geospatial application, for $c = 3$ it is necessary around 800,000 documents, whereas for $c = 200$, this amount drops to 15,000.

Another important metric from the create module is the time to create an index structure. Figure 11

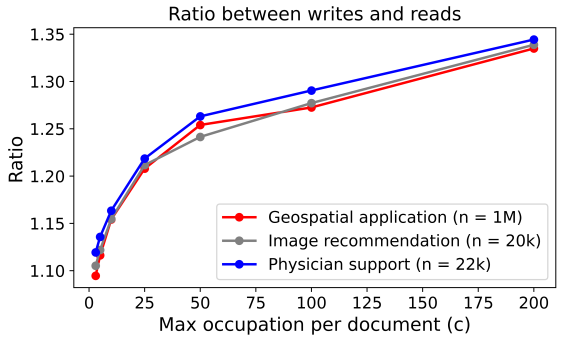

Figure 9: Ratio between reads and writes for creating the Slim-Tree on collection T_P.

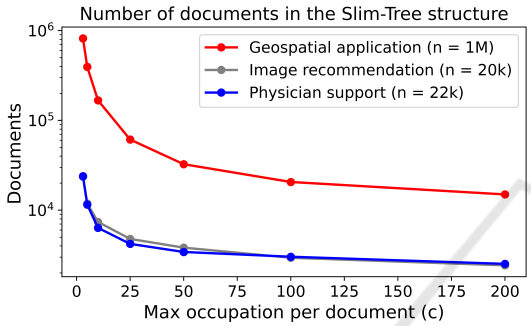

Figure 10: Number of documents on the collection T_P.

shows the total time required to create each Slim-Tree. As it can be seen, there is a minimum time that occurs at around $c = 10$ for all applications, and after that, the time increases significantly.

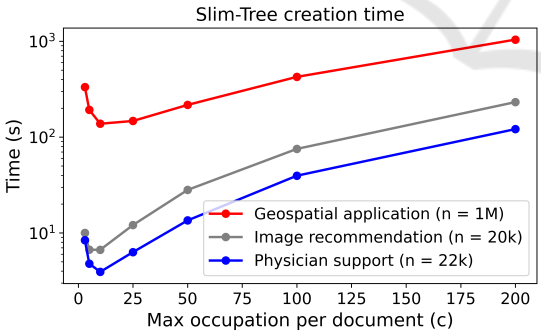

Figure 11: Time required to create a Slim-Tree.

4.2 The Similarity Query Module Costs

The main metric analyzed here is the financial cost of executing similarity queries. As discussed in Section 3.2, this cost is related to reading documents in the Slim-tree structure and then reading the documents that satisfy the query parameters on collection T_S.

We compare query executions asking the Similarity query module for the same queries executed through sequential scans on the entire T_S collection and using the index structure. Figure 12 shows the costs associated with the range (bottom row) and the *kNN* queries (top row), varying the number of elements per node as $c \in \{3, 25, 200\}$. For *kNN* queries, k varies from 1 to 100 (which covers the most frequent queries). For range queries, the radius varies from zero to the average radius obtained by the corresponding *kNN* query with $k = 100$.

As it can be seen, for small range and k thresholds, the Similarity-Slim extension (with $c = 200$) can reduce the similarity query cost by around $2,800$ times for the geospatial application, 130 times for image recommendation and 260 times on physician diagnosis support system, and it essentially keeps the same order of magnitude for the cost reduction as these thresholds increase. To help understand the reason why this cost reduction is achieved, Table 3 shows the average number of document reads in each collection for a *kNN* query (for $k = 5$) on the geospatial application: even with $15,000$ documents in collection T_P for $c = 200$ (Figure 10) and $1,000,000$ documents in collection T_S, only around 357.28 documents reads in collection T_P and 5 documents reads in collection T_s were required to perform the query.

Table 3: Documents reads in a *kNN* ($k = 5$) Query on a geospatial application ($n = 1M$).

Method	Reads from Tp	Reads from Ts
Sequencial scan	0	1,000,000
Slim-Tree (c=3)	16,584.82	5
Slim-Tree (c=25)	1,617.08	5
Slim-Tree (c=200)	357.28	5

The experiments show that the query cost decreases with increasing c. As discussed in Section 2.3, a Firestore document can store up to 1 *Mbytes* of data, so we can increase c until a document in T_P reaches this limit. However, to assist in defining a default value for c, we looked at the impact of c on common queries. Figure 13 shows the cost of *kNN* queries ($k = 5$) with varying c. As can be seen, after $c = 100$, there is only a marginal cost reduction for every evaluated application. Thus, we set $c = 100$ as the default. Furthermore, Figure 14 and Figure 15 show the impact of c in the query time and in the number of similarity calculations, respectively: this recommended default allows for reduction of the query time over every application.

We also compared the Similarity-Slim extension with sequentially scanning the full T_S collection regarding the total time required to execute each query.

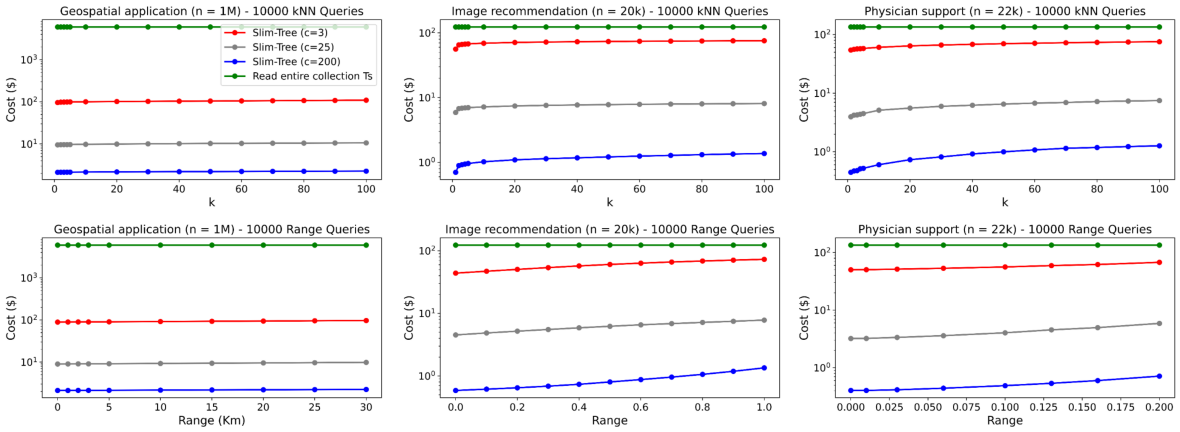

Figure 12: Financial cost of similarity queries varying the number of elements per node (c).

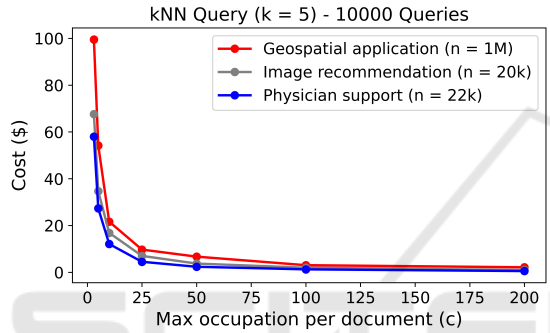

Figure 13: Impact of c in a *kNN* query cost.

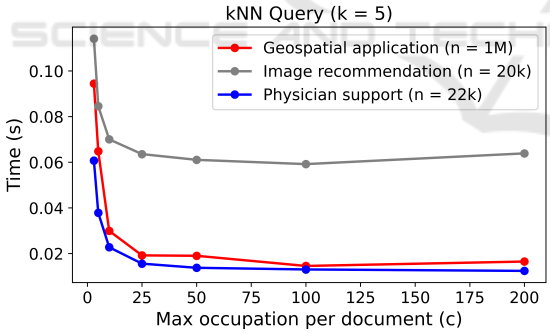

Figure 14: Impact of c in a *kNN* query time.

The experiments revealed that the extension could also accelerate the queries: it happens because, for similarity queries, the execution time is strongly related to the number of similarity calculations that must be executed, and a MAM targets to reduce them.

Figure 16 shows an example of the query time required by a *kNN* query with $k = 5$ for each application. As it can be seen, in addition to the financial cost reduction, a query executed in a Slim-Tree with $c = 100$ is 86 times faster in the geospatial application, 1.2 times faster on image recommendation, and 2 times faster on physician support, always returning

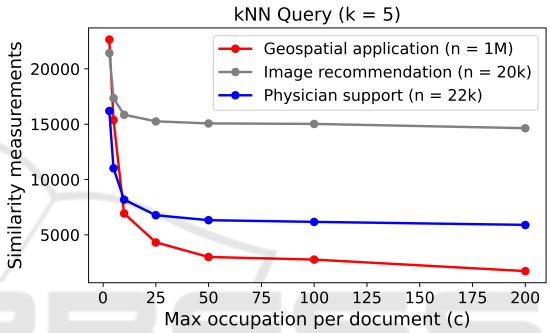

Figure 15: Impact of c in the number of similarity calculations in a *kNN* query.

the same result (the query answer is exact). In this example, the query execution took 360 times less similarity calculations on the geospatial application, 1.4 times less on image recommendation, and 3.6 times less on physician support.

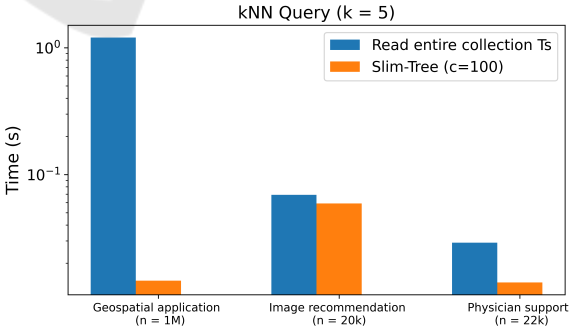

Figure 16: Time to perform the queries.

5 CONCLUSION

This paper presented Similarity-Slim, an extension for NoSQL document stores aiming at reducing the financial cost of performing similarity queries over

document collections in cloud-based data stores. It uses a Metric Access Method integrated into the data store's resources to reduce both the financial cost and the total query time of the similarity queries. The fundamental concepts presented can be applied to any metric domain whose datasets are described by documents stored in document stores, although in this paper, we evaluated its applicability using the Google Cloud Firestore as the case study. Regarding financial costs, the experiments showed that the extension always reduces the expenses of similarity queries. In fact, depending on the cardinality and dimensionality of the data, the extension was able to reduce the cost by up to $2,800$ times for small *range* and k.

We foresee that Similarity-Slim is a valuable resource to help make similarity queries more popular and accessible in NoSQL cloud-based systems and more specifically, in mobile and web app applications that use Firestore as data stores. In this work we evaluated Geo-spatial applications, recommendation systems, and physician diagnosis support systems as case studies, confirming that all of them can benefit from the concepts presented.

The core of the proposed extension consists of employing a successful existing indexing structure, originally developed to perform similarity queries in RDBMSs, now retooled to assist in obtaining cheaper storage and retrieval of documents in a NoSQL store. As the results obtained were very good, they provide support for us to explore the extension for other NoSQL databases, like MongoDB, to develop other types and variants of similarity queries, and undertake the development of a new, more refined MAM, specifically developed to further reduce the number of documents that need to be read when answering similarity queries over document stores.

ACKNOWLEDGEMENTS

We thank the support from the São Paulo Research Foundation (FAPESP, grant 2016/17078-0), the National Council for Scientific and Technological Development (CNPq), and the Coordination for Higher Education Personnel Improvement (CAPES).

REFERENCES

Barioni, M. C. N., dos Santos Kaster, D., Razente, H. L., Traina, A. J., and Júnior, C. T. (2011). Querying multimedia data by similarity in relational dbms. In *Advanced database query systems: techniques, applications and technologies*, pages 323–359. IGI Global.

Barioni, M. C. N., Razente, H., Traina, A., and Traina Jr, C. (2006). Siren: A similarity retrieval engine for complex data. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1155–1158.

Cazzolato, M. T., Scabora, L. C., Zabot, G. F., Gutierrez, M. A., Jr., C. T., and Traina, A. J. M. (2022). Featset+: Visual features extracted from public image datasets. *Journal of Information and Data Management (JIDM)*, 13(1).

Chen, L., Gao, Y., Song, X., Li, Z., Zhu, Y., Miao, X., and Jensen, C. S. (2022). Indexing metric spaces for exact similarity search. *ACM Computing Surveys*, 55(6):1–39.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, volume 97, pages 426–435.

Cong, G. and Jensen, C. S. (2016). Querying geo-textual data: Spatial keyword queries and beyond. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2207–2212.

Coşkun, İ., Sertok, S., and Anbaroğlu, B. (2019). K-nearest neighbour query performance analyses on a large scale taxi dataset: Postgresql vs. mongodb. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42:1531–1538.

Damaiyanti, T. I., Imawan, A., Indikawati, F. I., Choi, Y.-H., and Kwon, J. (2017). A similarity query system for road traffic data based on a nosql document store. *Journal of Systems and Software*, 127:28–51.

Deza, E., Deza, M. M., Deza, M. M., and Deza, E. (2009). *Encyclopedia of distances*. Springer.

Gonçalves, H. C., Carniel, A. C., Vizinhos-PR-Brazil, D., and Carlos-SP-Brazil, S. (2021). Spatial data handling in nosql databases: a user-centric view. In *GeoInfo*, pages 167–178.

Google (2023a). Cloud firestore. https://firebase.google.com/products/firestore?hl=pt-br. Last checked on Dec 03, 2023.

Google (2023b). Cloud functions. https://firebase.google.com/docs/functions?hl=pt-br. Last checked on Dec 03, 2023.

Google (2023c). Firebase extension: Reverse image search with vertex ai. https://extensions.dev/extensions/googlecloud/storage-reverse-image-search. Last checked on Dec 03, 2023.

Google (2023d). Firebase extension: Search firestore with algolia. https://extensions.dev/extensions/algolia/firestore-algolia-search. Last checked on Dec 03, 2023.

Google (2023e). Firebase extension: Semantic search with vertex ai. https://extensions.dev/extensions/googlecloud/firestore-semantic-search. Last checked on Dec 03, 2023.

Google (2023f). Firebase extensions. https://firebase.google.com/products/extensions?hl=pt-br. Last checked on Dec 03, 2023.

Google (2023g). Firestore emulator. https://firebase.google.

com/docs/emulator-suite/connect_firestore?hl=pt-br. Last checked on Dec 03, 2023.

Google (2023h). Firestore pricing. https://cloud.google.com/firestore/pricing?hl=pt-br. Last checked on Dec 03, 2023.

Karras, A., Karras, C., Samoladas, D., Giotopoulos, K. C., and Sioutas, S. (2022). Query optimization in nosql databases using an enhanced localized r-tree index. In *International Conference on Information Integration and Web*, pages 391–398. Springer.

Kesavan, R., Gay, D., Thevessen, D., Shah, J., and Mohan, C. (2023). Firestore: The nosql serverless database for the application developer. In *39th IEEE International Conference on Data Engineering, ICDE 2023*, pages 3376–3388, Anaheim, CA, USA. IEEE.

Kim, T., Li, W., Behm, A., Cetindil, I., Vernica, R., Borkar, V., Carey, M. J., and Li, C. (2020). Similarity query support in big data management systems. *Information Systems*, 88:101455.

Kim, T., Li, W., Behm, A., Cetindil, I., Vernica, R., Borkar, V. R., Carey, M. J., and Li, C. (2018). Supporting similarity queries in apache asterixdb. In *EDBT*, pages 528–539.

Koutroumanis, N. and Doulkeridis, C. (2021). Scalable spatio-temporal indexing and querying over a document-oriented nosql store. In *EDBT*, pages 611–622.

Li, R., He, H., Wang, R., Ruan, S., Sui, Y., Bao, J., and Zheng, Y. (2020). Trajmesa: A distributed nosql storage engine for big trajectory data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 2002–2005.

Lu, W., Hou, J., Yan, Y., Zhang, M., Du, X., and Moscibroda, T. (2017). Msql: efficient similarity search in metric spaces using sql. *The VLDB Journal*, pages 3–26.

MongoDB (2023). Mongodb. https://www.mongodb.com/. Last checked on Dec 03, 2023.

Nesso, M. R., Cazzolato, M. T., Scabora, L. C., Oliveira, P. H., Spadon, G., de Souza, J. A., Oliveira, W. D., Chino, D. Y., Rodrigues, J. F., Traina, A. J., et al. (2018). Rafiki: Retrieval-based application for imaging and knowledge investigation. In *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*, pages 71–76. IEEE.

Niwattanakul, S., Singthongchai, J., Naenudorn, E., and Wanapu, S. (2013). Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384.

Qader, M. A., Cheng, S., and Hristidis, V. (2018). A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, page 551–566, Houston, TX, USA. Association for Computing Machinery.

Roussopoulos, N., Kelley, S., and Vincent, F. (1995). Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 71–79.

Shimomura, L. C., Oyamada, R. S., Vieira, M. R., and Kaster, D. S. (2021). A survey on graph-based methods for similarity searches in metric spaces. *Information Systems*, 95:101507.

The Apache Software foundation (2023). Apache asterixdb. https://asterixdb.apache.org/. Last checked on Dec 03, 2023.

Traina-Jr, C., Traina, A., Seeger, B., and Faloutsos, C. (2000). Slim-trees: High performance metric trees minimizing overlap between nodes. In *Advances in Database Technology—EDBT 2000: 7th International Conference on Extending Database Technology Konstanz, Germany, March 27–31, 2000 Proceedings*, pages 51–65. Springer.

Unxos GmbH (2023). Geonames: geographical database. https://www.geonames.org/. Last checked on Dec 03, 2023.

William Zaniboni Silva (2024). Similarity slim - database and image group (gbdi-usp) - source code. https://github.com/WilliamZaniboni/ICEIS-2024-Similarity-Slim-Python. Last checked on Feb 10, 2024.

Wilson, D. R. and Martinez, T. R. (1997). Improved heterogeneous distance functions. *Journal of artificial intelligence research*, 6:1–34.

Yan, K., Peng, Y., Sandfort, V., Bagheri, M., Lu, Z., and Summers, R. M. (2019). Holistic and comprehensive annotation of clinically significant findings on diverse ct images: learning from radiology reports and label ontology. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8523–8532.

Yan, K., Wang, X., Lu, L., and Summers, R. M. (2018). Deeplesion: automated mining of large-scale lesion annotations and universal lesion detection with deep learning. *Journal of medical imaging*, 5(3):036501–036501.

Zhang, D. and Lu, G. (2003). Evaluation of similarity measurement for image retrieval. In *International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003*, volume 2, pages 928–931 Vol.2.