applied sciences

MDPI

Article

A Hybrid Approach Combining R*-Tree and *k*-d Trees to Improve Linked Open Data Query Performance

Yuxiang Sun, Tianyi Zhao, Seulgi Yoon and Yongju Lee *

School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea; syx921120@gmail.com (Y.S.); zty950731@gmail.com (T.Z.); tmffml123@naver.com (S.Y.)
* Correspondence: yongju@knu.ac.kr; Tel.: +82-10-3532-5295

Abstract: Semantic Web has recently gained traction with the use of Linked Open Data (LOD) on the Web. Although numerous state-of-the-art methodologies, standards, and technologies are applicable to the LOD cloud, many issues persist. Because the LOD cloud is based on graph-based resource description framework (RDF) triples and the SPARQL query language, we cannot directly adopt traditional techniques employed for database management systems or distributed computing systems. This paper addresses how the LOD cloud can be efficiently organized, retrieved, and evaluated. We propose a novel hybrid approach that combines the index and live exploration approaches for improved LOD join query performance. Using a two-step index structure combining a disk-based 3D R*-tree with the extended multidimensional histogram and flash memory-based *k*-d trees, we can efficiently discover interlinked data distributed across multiple resources. Because this method rapidly prunes numerous false hits, the performance of join query processing is remarkably improved. We also propose a hot-cold segment identification algorithm to identify regions of high interest. The proposed method is compared with existing popular methods on real RDF datasets. Results indicate that our method outperforms the existing methods because it can quickly obtain target results by reducing unnecessary data scanning and reduce the amount of main memory required to load filtering results.

Keywords: LOD cloud; Big Data; multidimensional histogram; index structure; join query; RDF

Citation: Sun, Y.; Zhao, T.; Yoon, S.; Lee, Y. A Hybrid Approach Combining R*-Tree and *k*-d Trees to Improve Linked Open Data Query Performance. *Appl. Sci.* **2021**, *11*, 2405. https://doi.org/10.3390/app11052405

Academic Editors: Dariusz Mrozek and Vaidy Sunderam

Received: 17 January 2021
Accepted: 23 February 2021
Published: 8 March 2021

1. Introduction

The evolution of the Linked Open Data (LOD) cloud has made a strong wave of research approaches in Big Data [1]. The LOD cloud consists of 1269 machine-readable datasets with 16,201 links (as of May 2020). LOD refers to a set of best practices for publishing and interlinking structured data for access by both humans and machines. The core concept of LOD is that the general Web architecture can be used to share structured data on a global scale. Technically, the LOD cloud employs the resource description framework (RDF) for data interchange and SPARQL for querying. The uniform resource identification (URI) is used to link related data across multiple resources. Vocabularies and ontologies define concepts and relationships to organize knowledge bases. In the RDF dataset, data items are expressed in the form of subject, predicate, and object triples. Because RDF triples are modeled as graphs, we cannot directly adopt existing solutions from relational databases and XML technologies [2]. Therefore, new storage and searching techniques are required to improve LOD query performance.

To achieve the best storage and searching strategy for the LOD cloud, it is necessary to consider existing approaches. Currently, four alternative approaches are available. First, we can store independent data copies in a local repository, benefiting from convenient conditions for efficient query processing; we call this the local approach. Local approaches gather data from all known resources in advance, preprocess the combined data, and store the results in a local repository [3]. Using such a repository, this approach can provide

excellent query response time owing to a lack of network traffic. However, this approach has many shortcomings. First, retrieved results may not reflect the most recent data. Second, copying all data can be expensive, and the performance penalty may become excessively high as the dataset volume increases. There is a large amount of unnecessary data collection, processing, and storage. Third, clients can only use a fragment of the Web of data copied into the repository.

The second approach is based on accessing distributed data on the fly using a recursive URI lookup process; we call this the live exploration approach. This approach performs queries over multiple SPARQL endpoints offed by publishers for their LOD datasets [4]. This approach has several advantages, such as synchronizing copied data is not required, searching is more dynamic with up-to-date data, and new resources can be easily added without a time lag for indexing and integrating data. Moreover, this approach requires less storage. However, this approach may not guarantee that all publishers offer reliable SPARQL endpoints for their LOD datasets [4].

The third technique is the index approach. LOD index structures are quite similar to traditional database query processing techniques. Existing data summaries and approximation techniques can be adapted to develop an index structure for LOD queries. For instance, Umbrich et al. [5] considered multidimensional histograms (MDHs) as a summarizing index for LOD query processing. Harth et al. [3] proposed an approximate multidimensional index structure (i.e., QTree), which combines the MDH and R-trees [6]. In contrast to MDHs, where regions have a fixed size, QTree is a tree-based data structure where variable-size regions more accurately cover the content of resources [5]. Although MDHs provide an easy method for constructing and maintaining the index, it is considered a rather rough approximation. QTree presents a more accurate approximation than MDHs; however, QTree requires a high cost for building and maintaining the trees.

Finally, the fourth technique is the hybrid approach. This approach combines two storage and searching approaches. Consequently, the hybrid approach can conceivably obtain the advantages of both approaches without their respective drawbacks. For instance, Umbrich [7] proposed a hybrid query framework that offers fresh and fast query results by combining the local and live exploration approaches. This method employs a query planner that decides the parts of the query delegated to the local and remote engines. Lyden et al. [8] proposed a hybrid approach that combines link traversal and distributed query processing on the fly to retrieve relevant data within a fixed timeframe. The local approach offers the best query performance; however, the retrieved data might not be up-to-date. The live exploration approach provides recent data; however, the query execution is slower than the local approach because data must be transmitted via a network. The hybrid approach can address these problems. However, to the best of our knowledge, this approach cannot yet provide optimal solutions for complex join query processing. Table 1 briefly describes these four approaches.

Table 1. Storage and searching approaches.

	Local Approach	**Live Exploration Approach**	**Index Approach**	**Hybrid Approach**
Feature	Store collected data into a local repository	Query multiple SPARQL endpoints	Use summary and approximation indexes	Combine two storage and searching approaches
Advantage	Excellent response time	Dynamic with up-to-date data	Efficient query processing	Trade-off between two approaches
Disadvantage	Cannot reflect recent data	Slow response time	High maintenance cost	Bad join query performance
Related work	QUAD [9], RDF-3X [10], Hexastore [11], Matrix [12], TripleBit [13]	DARQ [14], SemWIQ [15], LiveExplorer [4], Lusail [16], IRISelection [17]	MDH [5], QTree [3], MIDAS-RDF [18], SameAsPrefixIndex [19]	HybridEngine [7], HybridQuery [8], H-Grid [20], MapReduce+RDF-3X [21]

We propose a novel hybrid approach that combines the index and live exploration approach. We develop a two-step index structure comprising an approximate disk-based index structure (i.e., 3D R*-tree) and flash memory-based *k*-d trees. Using this index structure, our approach can efficiently store and search the LOD cloud distributed across multiple resources. Furthermore, we propose a hot-cold segment identification algorithm to determine ways for relocating data to disk and flash memory. The contributions of this study are summarized:

- We present a new hybrid index structure, the two-step index structure, designed for efficient LOD join queries. In particular, we consider flash-based solid-state drives (SSDs) as excellent memory-based *k*-d trees.
- We propose an efficient join query algorithm based on the two-step index structure for various SPARQL query types and a hot-cold segment identification algorithm that determines regions of high interest.
- We evaluate our index structure through extensive experiments using benchmark LOD datasets. Experimental results show that our hybrid approach exhibits better retrieval performance than existing approaches.

The rest of this paper is organized as follows. In Section 2, we describe the research background. In Section 3, the new two-step hybrid index structure and hot-cold segment identification algorithm for efficient SPARQL query processing are proposed. In Section 4, the experimental evaluation is described. In Section 5, conclusions are drawn.

2. Background and Related Work

2.1. Overview of Linked Open Data

Spurred by efforts such as the LOD project [22], large amounts of semantic data are published in the RDF format in several diverse fields such as publishing, life sciences, social networking, internet of things (IOT), and healthcare. The LOD cloud now covers 1269 datasets from diverse domains. Especially, the emergence of LOD has been making an excellent revolution in the healthcare sector. For example, corona virus disease 19 (COVID-19) is an infectious disease caused by a newly discovered coronavirus. In the midst of a global pandemic, Big Data such as LOD are showing an outstanding capability to analyze complex interactions among groups of people and locations. Through the work such as sharing and analyzing available LOD in the world, a cure for COVID-19 will be discovered. Efficient use of the large-scale LOD could find to a way to divine how the virus is spreading and how the number of infections can be reduced. The processing of such Big RDF Data requires a scalable data management system that can efficiently store, index, and query RDF data.

RDF is a graph-based data model for the LOD cloud, and SPARQL is a standard query language for this data model. Assume **U**, **B**, and **L** are a set of all URIs, blank nodes, and literals, respectively. An RDF triple is a tuple t = (s, p, o) $\in$ (**U** $\cup$ **B**) $\times$ **U** $\times$ (**U** $\cup$ **B** $\cup$ **L**), where *s*, *p*, and *o* are the subject, predicate, and object, respectively. For instance, the triple (<http://dbpedia.org/person/Smith>, <http://xmlns.com/foaf/0.1/knows>, <http://university.edu/students/Lucy>, accessed on 17 January 2021) indicates "Smith knows Lucy." SPARQL is a graph-matching query language, and a SPARQL query is typically a combination of graph patterns. A basic graph pattern is a set of triple patterns, where a triple pattern is an RDF triple that can contain query variables (prefixed with '?') at the subject, predicate, and object positions. The main operation of SPARQL is matching these basic graph patterns.

Example 1: *The following SPARQL query asks for the projects in which Smith's friends participated. This query comprises two triple patterns joined by variable ?f:*

```
PREFIX   foaf:http://xmlns.com/foaf/0.1/ (accessed on 17 January 2021)
PREFIX   user:http://dbpedia.org/person/ (accessed on 17 January 2021)
SELECT   ?n WHERE {
         user:Smith  foaf:knows    ?f .
         ?f          foaf:project  ?n }
```

Because SPARQL statements can be represented as graphs, we can transform SPARQL into a query graph to perform query processing. Based on the query graph structure, SPARQL queries can be divided into five different join types [23]—star, chain, directed cycle, complex, and tree—as shown in Figure 1. Our experiments include all join query types based on the real benchmark dataset.

- Star queries are a set of triples formed using the same subject or object variable (subject = subject or object = object). Usually, we consider only subject-subject joins (i.e., all triples have the same subject).
- Chain and directed cycle queries are triple patterns in which the subject and object variables are the same (subject = object) (i.e., the object of the triple is the subject of the next triple).
- Complex queries are a combination of star and chain queries.
- Tree queries contain subject-subject and subject-object joins and some more complex queries.

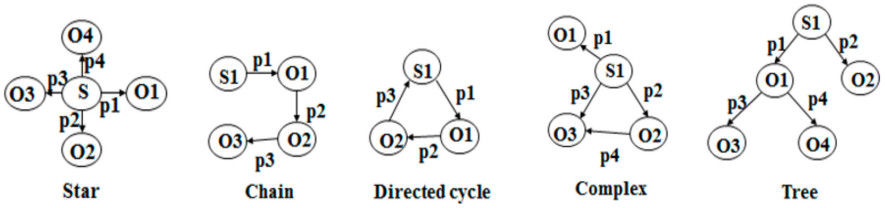


Figure 1. SPARQL query types.

Critical problems in the LOD cloud are the massive number of LOD datasets and potentially very large numbers of joins required to answer the queries. Thus, careful consideration must be provided to the system design to implement an efficient physical structure with suitable indexes and support join queries. Typically, LOD index structures only need to consider the found relevant resources. However, there is no guarantee that the resources actually provide the RDF triple that we were originally looking for (i.e., false positives can occur) [5]. This is because the minimum bounding box (MBB) container is considered a rather inaccurate approximation. Because of the rough approximation, the candidates may contain a number of false hits not fulfilling the query condition. Furthermore, all candidates must be transmitted into the refinement phase, even if they contain false hits. In case of large numbers of resources, searching a desired result can be very expensive because exact calculations are required for accessing several unreliable resources over the LOD cloud.

2.2. Hybrid Storage Structure

The traditional high-performance database system is mainly based on hard disk drives (HDDs). Currently, HDD is still the main device used in storage systems. Although its capacity can be rapidly increased by increasing the number of disk slices and disk partitions, it still rapidly develops according to Moore's Law. Affected by this, the random-access performance of HDDs has not been significantly improved, which is a bottleneck. SSD is a new type of storage device that uses integrated circuits as a memory for the persistent storage of data. Compared with HDD, SSD yields outstanding read and write performance. However, SSD also has shortcomings, such as low capacity and high cost. Currently, no storage system comprising a single storage medium can meet the requirements of low cost and high query performance. Capacity, cost, sequential performance, random performance, and device life can together be considered as the primary indicators of the comprehensive level of the storage system. As shown in Figure 2, the range of requirements covered by a single storage medium is limited. If two storage media are integrated to form a hybrid storage system, such a system could conceivably utilize the strengths and avoid the weaknesses of each medium.

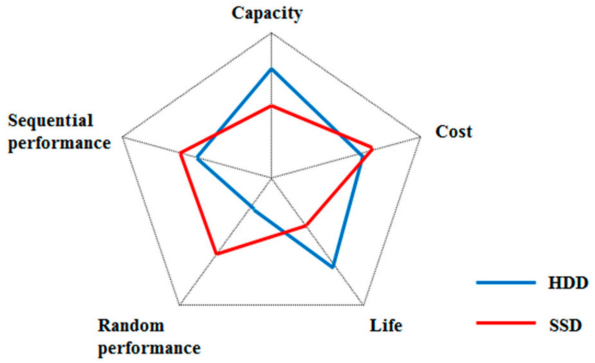


Figure 2. Comparison between solid-state drive (SSD) and hard disk drive (HDD).

To improve the storage and query efficiency, in addition to considering the storage device, it is essential to effectively access data. Therefore, an efficient storage system must guarantee excellent retrieval performance. Indexes are special-purpose data structures designed to accelerate data retrieval. Many researchers have conducted extensive research on indexes with respect to HDDs. Moreover, numerous methods have been suggested, most of which are based on popular B-trees [24], R-trees [6], linear [25], and scalable hashing [26]. These data structures can be directly used with SSDs. However, their performance is unsatisfactory because of several significant differences between integrated circuits and magnetic disks, such as asymmetric read-write latencies, out-of-place updates that must be erased before writing, and limited cell lifetime.

2.3. Related Work

The goal of this subsection is to briefly present a set of RDF storage engines designed for large-scale Semantic Web applications. It should be stressed that the related work presented here is not exhaustive, but we make a detailed quantitative analysis of system performance with these engines in Section 4.

QUAD [9] uses quads in the form of <subject, predicate, object, context>. It stores RDF data persistently by using six B+ tree indices. The indices cover all the 16 possible access patterns of quads. This representation allows the fast retrieval of all triple access patterns. To speed up keyword queries, the lexicon keeps an inverted index on string literals to allow fast full-text searches. To retrieve any access pattern with a single index lookup, QUAD sacrifices storage space and insertion speed for query performance since each triple is encoded in the dictionary six times. Thus, it seems to be primarily geared for simple lookup operations with limited support for joins; it lacks DBMS-style query optimization (e.g., do not consider any join-order optimizations).

DARQ [14] is a query engine for federated SPARQL queries. It supports transparent query access to multiple, distributed SPARQL endpoints as if querying a single RDF graph. In DARQ, the original SPARQL query is decomposed into several subqueries, where each subquery is sent to its relevant SPARQL endpoints. The results of subqueries are then joined together to answer the original SPARQL query. A service description language enables the query engine to decompose a query into subqueries. Furthermore, DARQ discusses the use of semi-joins to compute a join between intermediate results at the control site and SPARQL endpoints. The disadvantages are the latency of query execution since live exploration algorithms seek SPARQL endpoints, and repeated data retrieval from each endpoint introduces significant latency. They also have limited possibility for parallelization.

MIDAS-RDF [18] is a distributed RDF store that is built on top of a distributed multidimensional index structure. Through the multidimensional index, it is able to support an efficient range search over the overlay network. It features the fast retrieval of RDF triples satisfying various pattern queries by translating them into multidimensional range queries, which can be processed by the underlying index in hops logarithmic to the

number of peers. MIDAS-RDF achieves better performance by using a labeling scheme to handle expensive transitive closure computations efficiently. It also implements reasoning services for the RDFS entailment regime. However, MIDAS-RDF is limited to sharing, querying, and synchronizing distributed RDF repositories.

3. Hybrid Index System

In this section, we propose an extended multidimensional histogram and a two-step index structure for improving LOD query performance. Algorithms of the hot-cold segment identification and two-step join query processing based on our index structure are discussed.

3.1. Extended Multidimensional Histogram

We adapt the MDH technique for LOD storage and searching. Our histogram, called MDH*, aims to support efficient join query processing without significant storage demand. To achieve scalable query processing, we design a compact storage structure and minimize the number of indexes used in the query execution. As a running example, we extract sample data from a small portion of the RDF file. Figure 3 shows these sample data.

(Smith, knows, Lucy), (Smith, knows, Mary), (Smith, knows, Julia),
(Brown, knows, Holis), (Brown, knows, Tina), (Maya, knows, Lucy),
(Maya, knows, Mary), (Maya, knows, Tina), (Lucy, project, LinkedData),
(Lucy, project, SemanticWeb), (Julia, project, Ontology),
(Julia, project, LinkedData), (Julia, project, SemanticWeb),
(Holis, project, Ontology), (Holis, project, DataMining),
(Tina, project, LinkedData), (Tina, project, BigData)

Figure 3. Sample data used for the running example.

In MDH*, we adopt a dictionary compression method [13] to reduce the redundancy of long common URI prefixes. This method divides a URI into prefix and suffix parts using the last separator "/". Moreover, instead of storing the prefix and suffix parts as a string, all strings are replaced with unique numerical codes. For instance, Smith would be represented by a URL such as "http://dbpedia.org/person/Smith" (accessed on 17 January 2021) in a real-world RDF graph; storing compressed numbers instead of this string can save large spaces. Here, we provide a brief introduction of the MDH concept [5]. The first step in building the MDH involves transforming the RDF triples into points in a three-dimensional space by applying hash functions (e.g., hashCode() function in Java) to the individual components of the RDF triples. Next, the three-dimensional space is partitioned into disjoint regions, each defining a so-called bucket. Each bucket contains entries of all URIs whose data include RDF triples in the corresponding region. Given a triple pattern for an RDF query, a lookup entails computing the corresponding triple points by applying the same hash function and retrieving buckets responsible for the obtained triple points. Any URI relevant to the triple pattern can only be contained in buckets.

Similarly, the first step of building the MDH* involves transforming the RDF triples into compressed numbers using Java's hashCode() function. In this case, these numbers are points in the n-dimensional data space, whose coordinates correspond to three-dimensional cubes for (s, p, o). The coordinates are inserted sequentially and aggregated into regions. Each region contains a list of these compressed triple numbers. The triple numbers in the list are added with two special count frequencies, rather than ordinary triple numbers, to improve the join query performance. The count frequencies specify α and β, where α indicates the number of subjects in which o occurs as subjects in the LOD cloud; similarly, β indicates the number of objects in which s occurs as objects. We observe that a number of triples in the LOD cloud are used as subjects of one triple and objects of another triple. Yuan et al. [13] demonstrated that more than 57% of LOD subjects are also objects.

Definition 1: *Let (t, f) be an extended RDF tuple with frequency f, where t is the compressed triple numbers (x, y, z) and f is the count frequency (α, β). Then, the extended RDF tuple ((x, y, z), (α, β)) equals the quintuple (x, y, z, α, β).*

Considering the running example presented in Figure 3, the converted compression numbers are represented in Figure 4a. We can then obtain the compressed triple numbers for each RDF triple (e.g., (Smith, knows, Lucy) → (11, 12, 13)). The count frequencies are initialized (0, 0) if *o* and *s* have not appeared in *s* and *o*, respectively. Otherwise, when *o* or *s* appears in *s* or *o*, respectively, the values are incremented by 1. Figure 4b shows the completed extended RDF tuples in our running example after adding the count frequencies.

Smith=11, Brown=31, Maya=1, Lucy=13, Julia=4, Holis=33, Tina=23, Mary=3, knows=12, project=32, SemanticWeb=38, Ontology=6, BigData=36, DataMining=26, LinkedData=17

(a) Converted compression numbers

(11, 12, 13, 2, 0),	(11, 12, 3, 0, 0),	(11, 12, 4, 3, 0),	(31, 12, 33, 2, 0),
(31, 12, 23, 2, 0),	(1, 12, 13, 2, 0),	(1, 12, 3, 0, 0),	(1, 12, 23, 2, 0),
(13, 32, 17, 0, 0),	(13, 32, 38, 0, 0),	(4, 32, 6, 0, 0),	(4, 32, 17, 0, 0)
(4, 32, 38, 0, 0)	(33, 32, 6, 0, 0)	(33, 32, 26, 0, 0)	(23, 32, 36, 0, 0)
(23, 32, 17, 0, 0)			

(b) Extended RDF tuples

Figure 4. Converted compression numbers and extended resource description framework (RDF) tuples.

Because calculating the count frequencies would be expensive, counting and inserting the frequencies can be performed as batch processing after compressed triple numbers are established. Figure 5 schematically depicts the extended RDF tuples stored in a multidimensional histogram, i.e., MDH*. When a triple pattern for the RDF query is given, the main operation is to determine relevant extended RDF tuples that satisfy the query. By looking up these tuples in MDH*, we can obtain a set of candidate results that potentially provide relevant resources. In Section 3.4, we explain how SPARQL join queries can be efficiently performed using MDH*.

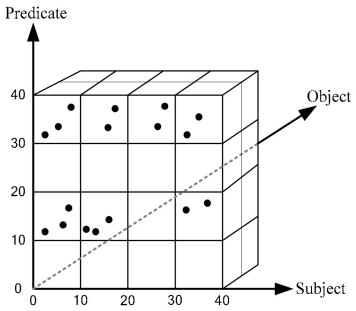


Figure 5. Extended RDF tuples in the three-dimensional MDH*.

3.2. Two-Step Index Structure

The success of the MDH* method depends on its ability to rapidly narrow down a set of relevant resources that are affected by the SPARQL queries. To decide which resources are relevant for a particular query, we need an efficient index structure that can organize the set of resources. A number of index structures for LOD have been proposed, including QTree, R-tree, *k*-d tree, and SameAsPrefixIndex. However, these structures are deemed unsuitable for organizing LOD datasets. Because of rough approximation, the candidate results contain large numbers of false hits that do not fulfill the query condition. All candidates must be transmitted into the refinement processing, even if they contain false hits. The refinement processing in the main memory is expensive, particularly if the

number of candidates is large, because time-consuming internet access is required for many unreliable resources.

In this section, we do not aim to discuss in detail which index structure is most suitable for organizing LOD datasets. Instead, we propose an extension of the existing index structure. Because the cost of developing a new index structure can be more expensive than the cost of simply extending an existing one, adding new features to the existing index structure is an excellent alternative. We select 3D R*-tree [27] for our index structure. R*-tree, the most popular variant of R-tree, is originally a multidimensional index structure for spatial data. Figure 6 depicts our two-step index structure. This figure is distinguished into two parts. The first level indicates the multidimensional index structure for organizing MDH*, which is a straightforward modification of the 3D R*-tree. The second level shows a set of *k*-d trees, which are used for refinement processing. Using this two-phase mechanism, we can completely filter irrelevant data.

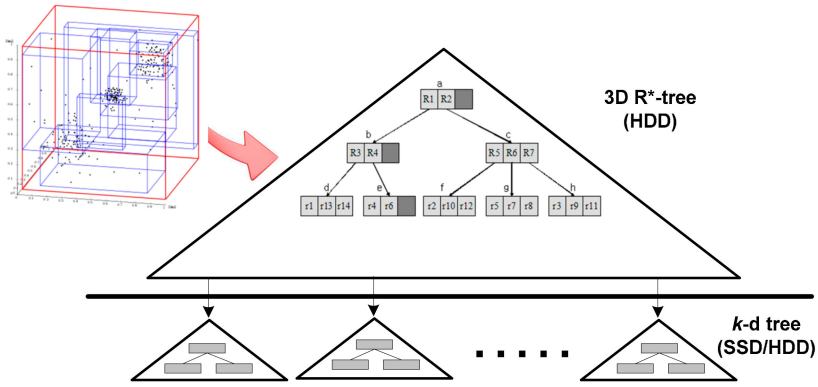


Figure 6. Two-step index structure based on minimum bounding box (MBB) approximation and MDH*.

The 3D R*-tree comprises leaf and nonleaf nodes. A leaf node contains entries of the form (id, region), where id refers to a *k*-d tree in the database and region is the MBB of that tree. A nonleaf node is built by grouping MBBs at the lower level. Each *k*-d tree node maintains a list of (uri, tuple) pairs, where uri is a pointer to the address where the object actually resides and tuple is an extended RDF tuple. In our case, the extended RDF tuples comprise points in the *n*-dimensional data space, which are represented by compressed hash values and count frequencies. We can significantly reduce the memory pressure by replacing long string literals with hash values. The hash values are inserted sequentially and aggregated into the *k*-d trees. The 3D R*-tree is well suited for disk use because it is a broad shallow tree and requires few node accesses to retrieve a value. From our experiments in Section 4.2, we find that the 3D R*-tree stored in SSD does not have much performance effect. Thus, all R*-tree nodes are stored in HDD because the cost of SSD is higher than that of HDD.

Each list of (uri, tuple) pairs is maintained on a designated *k*-d tree, which is completely transferred into the main memory when the refinement processing requires the corresponding list. The *k*-d tree is suitable for memory usage owing to its excellent storage utilization capacity (the pointer to data ratio is small and leaf nodes hold only data items), rapid searching (few nodes are retrieved using a binary search), and fast updating. However, existing Big Data systems either suffer from large memory space requirements or incur huge computational overhead. To improve the overall performance of the storage system, we divide the data into hot and cold. The frequently accessed data (i.e., hot data) are stored in SSDs, and the less frequently accessed data (i.e., cold data) are stored in HDDs. To distinguish hot and cold data more effectively, we use access history to train the decision tree model [28]. The trained model is used to flag the data and balance the hot/cold data ratio in SSD and HDD. Compared with MBF (Multiple Bloom Filter) [29] and HotDataTrap [30], our decision tree method is more intelligent and flexible.

3.3. Hot-Cold Segment Identification Algorithm

We propose a hot-cold segment identification algorithm to determine ways for reallocating data in SSD and HDD. We use the decision tree to identify data. The training data collection process for the decision tree is explained. When users access the data, RDF triples are stored in the decision table (see Figure 7). S, P, and O attributes are the hash values of the RDF triple elements. The RECENT attribute is a bit for determining whether the data have recently been accessed. Our algorithm applies the aging mechanism of HotDataTrap and its recency capturing mechanism sets the corresponding RECENT bit to 1 if any data are accessed. The COUNT attribute refers to the number of times the data have been accessed. Finally, the TYPE attribute is a bit for storing data based on whether they are hot or cold. If the RECENT value is 1 and the COUNT value is more than the threshold, the data are identified as hot data.

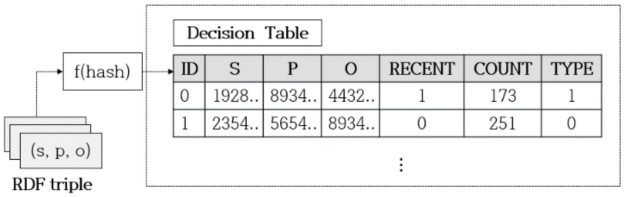


Figure 7. Decision table for training data.

To identify our training data, the decision tree uses an information gain approach for determining a suitable property. We select the attribute with the highest information gain as the test attribute. If TYPE is selected as the property for test, the required information entropy is expressed as

$$\mathrm{E}(Type) = \frac{Hot(\#)}{All(\#)} log_2 \frac{1}{\frac{Hot(\#)}{All(\#)}} + \frac{Cold(\#)}{All(\#)} log_2 \frac{1}{\frac{Cold(\#)}{All(\#)}} \quad (1)$$

where *Hot*(#), *All*(#), and *Cold*(#) denote the number of hot data, all data, and cold data in the decision table, respectively. The TYPE information entropy for RECENT is expressed as

$$\mathrm{E}(Type|Recent) = Prob(1) * \mathrm{E}(Type|1) + Prob(0) * \mathrm{E}(Type|0) \quad (2)$$

where *Prob*(1) and *Prob*(0) indicate the probabilities of one and zero. Thus, the information gain of RECENT can be obtained using Equation (1) minus Equation (2). In this way, the order of the attributes is determined. The decision tree then identifies the test data not accessed by the user and determines the accuracy to prove that the result is reliable. The data identified by the decision tree are relocated to the appropriate storage. The R*-tree buckets comprising hot data are classified as hot segments, and the remaining buckets are classified as cold segments. The hot segments are indexed into *k*-d trees in SSD, and cold segments are indexed into *k*-d trees in HDD. Algorithm 1 shows the proposed hot-cold segment identification algorithm.

The disadvantages of SSD include erase-before-write and limited life cycle. Because SSD uses the out-of-place update structure, it must first erase the old data to write new data. Unnecessary erase and write operations lead to wear-off of the cells of SSD, thereby reducing SSD lifetime. To reduce unnecessary write operations, we collect data until they reach the SSD page size and then execute the write operations. This technique removes redundant write operations and reduces the occurrence of problems related to SSD.

Algorithm 1. Hot-cold segment identification.

```
// Collecting training data for the decision tree
While (iterate time < predetermined time)
    If user accesses data Then
        RECENT = 1
        COUNT += 1
    End If
End While
While (there exist data in the decision table)
    If RECENT == 1 and COUNT >= threshold Then
        TYPE = 1 // Data are identified as hot data
    End If
End While
// Training decision tree
dTree = DecisionTreeClassifier(max_depth = 3) // Create decision tree
dTree.fit(train_data, train_label) // Train decision tree
result = dTree.predict(test_data) // Identify test data not accessed by the user.
Identified data are relocated to HDD and SSD.
```

3.4. Two-Step SPARQL Query Processing

Two types of queries that we concentrate on are the single and join triple query patterns. Because the processing of the single triple query pattern is straightforward, we focus on the join triple query pattern, which is expressed as a conjunction of multiple triple patterns. There are eight triple patterns in the SPARQL queries: (s, p, o), (s, p, ?o), (?s, p, o), (s, ?p, o), (s, ?p, ?o), (?s, ?p, o), (?s, p, ?o), and (?s, ?p, ?o) [13]. Because (?s, ?p, ?o) requires a full scan and the matching number of (s, p, o) is either 0 or 1, only the execution procedures regarding the other six triple patterns need to be evaluated.

Due to page length limitations, we only explain the evaluation of (s, p, ?o) here. The processing of this pattern comprises two steps: the filtering and refinement phases. The filtering phase prunes the search space using the 3D R*-tree. Using this tree, we determine all candidate MBBs that possibly provide the result. The triple pattern (s, p, ?o) is first transformed into coordinates in the three-dimensional space by applying the same hash function used for building the MDH*. Because the triple pattern includes a query variable (prefixed with ?) at the object position, this processing is treated as a line rather than a point (note that (?s, p, ?o) is treated as a plane). Using this query line, we can locate all MBBs overlapping the line in the 3D R*-tree. After identifying all candidate MBBs, we can refine the immediate result using the *k*-d trees.

For the immediate result obtained in the filtering phase, the refinement phase precisely evaluates the query condition. Using our 3D R*-tree, we can quickly select MBBs that contain all possible extended RDF tuples matching the join triple query pattern. Considering Example 1 of the (s, p, ?o) $\bowtie$ (?s, p, ?o) chain query type, the join triple query pattern combines extended RDF tuples using two or more candidate results. Although join processing has been extensively studied in the literature, algorithms designed for traditional join processing can hardly be used for the SPARQL join without modifications. Only the nested-loop join algorithm can be used without any modification. Thus, the nested-loop join algorithm serves as the starting point.

Algorithm 2 shows a detailed illustration of our MDH*-based join processing algorithm for the chain query type. It is a revised version of the traditional nested-loop join algorithm. In line 1, we determine whether a query is a possible SPARQL triple pattern. In line 4, we obtain a filtering result by joining the previous and current results using MBB. The input of our refinement phase is a set of pairs of relevant extended RDF tuples in the *k*-d trees. In the join procedure of line 13, two input sets (X and Y) are compared in the inner loop using equijoin techniques to determine the matching between the sets. If $\alpha \neq 0$, lines 17–20 are performed. This process first checks whether the two input sets satisfy the join condition. If so, these two tuples are appended to the final result and α is decremented by 1. This procedure is repeated until α becomes 0. Therefore, our algorithm can skip unnecessary operations that clearly do not fulfill the join condition. After eliminating these

false hits, a recursive URI lookup process [4] is performed using link traversal techniques. A similar algorithm is applied to β for the (?s, p, o) ⋈ (?s, p, ?o) query pattern (e.g., select ?n where {?f foaf:lastName Lucy. ?n foaf:knows ?f}).

Algorithm 2. MDH*-based join processing.

```
If Q is in SPARQL triple patterns // Q: query
    For each pattern i in Q
        If FilterPhase(i) != null Then
            FilterPhase(i) = OVERLAP(FilterPhase(i-1), FilterPhase(i))
        Else break
    End For
    For each pattern i in Q
        If RefinePhase(i) != null Then
            RefinePhase(i) = JOIN(RefinePhase(i-1), RefinePhase(i))
        Else break
    End For
End If
Procedure JOIN(X, Y) // X, Y: two k-d tree input sets
For each tuple x in k-d tree X
    For each tuple y in k-d tree Y
        If α ≠ 0 (or β ≠ 0) Then
            If x and y satisfy the join condition, Then
                x and y tuples are added to the result
                α (or β) is decremented by 1
            Else
        Else break
    End For
End For
```

Example 1 shows a SPARQL query expressing the join between two triple patterns. In traditional join techniques, after receiving a SPARQL query, a query engine usually performs a nested-loop join algorithm without considering the count frequencies. Figure 8 shows two input sets of the nested-loop join algorithm using our running example. The input sets in this figure are shaded. The join cost will be approximately $3 \times 9 = 27$ tuple comparisons. If the query is complex and the dataset is very large, the cost will significantly increase. The main concept of our method is decreasing the join cost by considering the count frequencies; if $\alpha = 0$, tuples need not be compared. Hence, two tuples (11, 12, 13, 2, 0) and (11, 12, 4, 3, 0) are selected as the input set of the join procedure. Furthermore, our join algorithm is only repeated within the α count. Thus, the join cost is reduced to $2 + 0 + 5 = 7$ tuple comparisons. This method can achieve up to 74% improvement for the join query.

(11, 12, 13, 2, 0),	(11, 12, 3, 0, 0),	(11, 12, 4, 3, 0),	(31, 12, 33, 2, 0),
(31, 12, 23, 2, 0),	(1, 12, 13, 2, 0),	(1, 12, 3, 0, 0),	(1, 12, 23, 2, 0),
(13, 32, 17, 0, 0),	(13, 32, 38, 0, 0),	(4, 32, 6, 0, 0),	(4, 32, 17, 0, 0)
(4, 32, 38, 0, 0)	(33, 32, 6, 0, 0)	(33, 32, 26, 0, 0)	(23, 32, 36, 0, 0)
(23, 32, 17, 0, 0)			

Figure 8. Two input sets of the nested-loop join algorithm.

4. Experimental Evaluation

In this section, our hybrid method is compared with existing popular methods on benchmark LOD datasets. We also measure the overall performance of the hot-cold segment identification algorithm using a real LOD dataset.

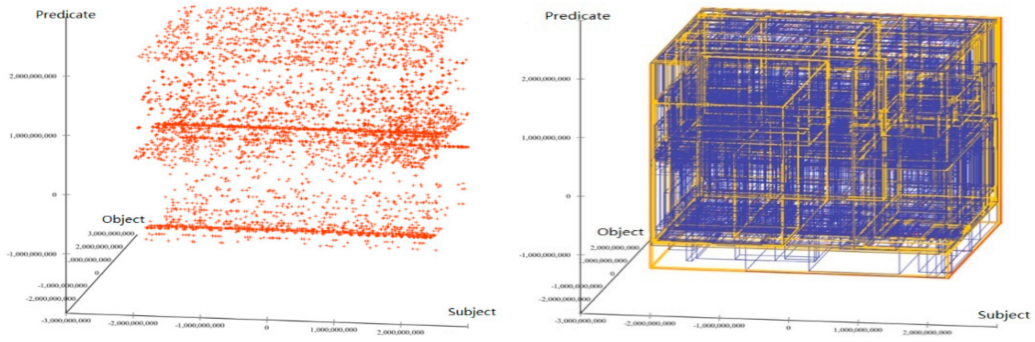
4.1. Join Query Performance and Storage Amount

In our experiments, we compared our method with other popular existing methods. Our objective is to show that we can achieve excellent join query performance with a small amount of storage. In the experiments, our hybrid approach, referred to as HYBRID, was compared with QUAD [9], DARQ [14], and MIDAS-RDF (referred to as MIDAS) [18].

QUAD, DARQ, and MIDAS are existing local, live exploration, and index approaches, respectively. This evaluation is conducted on a server system with a 3.6-GHz Intel i7 CPU and 32 GB memory. All programs were written in the Java language on a server running Windows 10. In this work, we performed two sets of experiments; first, we measured the join query performance, and then we measured the storage amount.

We used the Lehigh University Benchmark (LUBM) dataset [31] to obtain realistic results of the performance of large-scale RDF storage structures. The LUBM dataset is the most widely used benchmark dataset in the Semantic Web community. It contains 230,061 triples, 38,334 subjects, 17 predicates, and 29,635 objects. The size of the entire dataset is 36.7 MB. The original version of the LUBM queries has not been updated since 2009. With an increase in the query complexity, the older version cannot support all join query types. To address this issue, we modify the original version to support all join query types. We first design 15 queries from Q1 to Q15. Then, we enumerate all join query types and compare their join query performance. The SPARQL join query types are star (Q1, Q3, Q4, Q5, Q11, Q13), chain (Q6, Q10), directed cycle (Q2), cycle (Q15), tree (Q9), and complex (Q7, Q8, Q12) types.

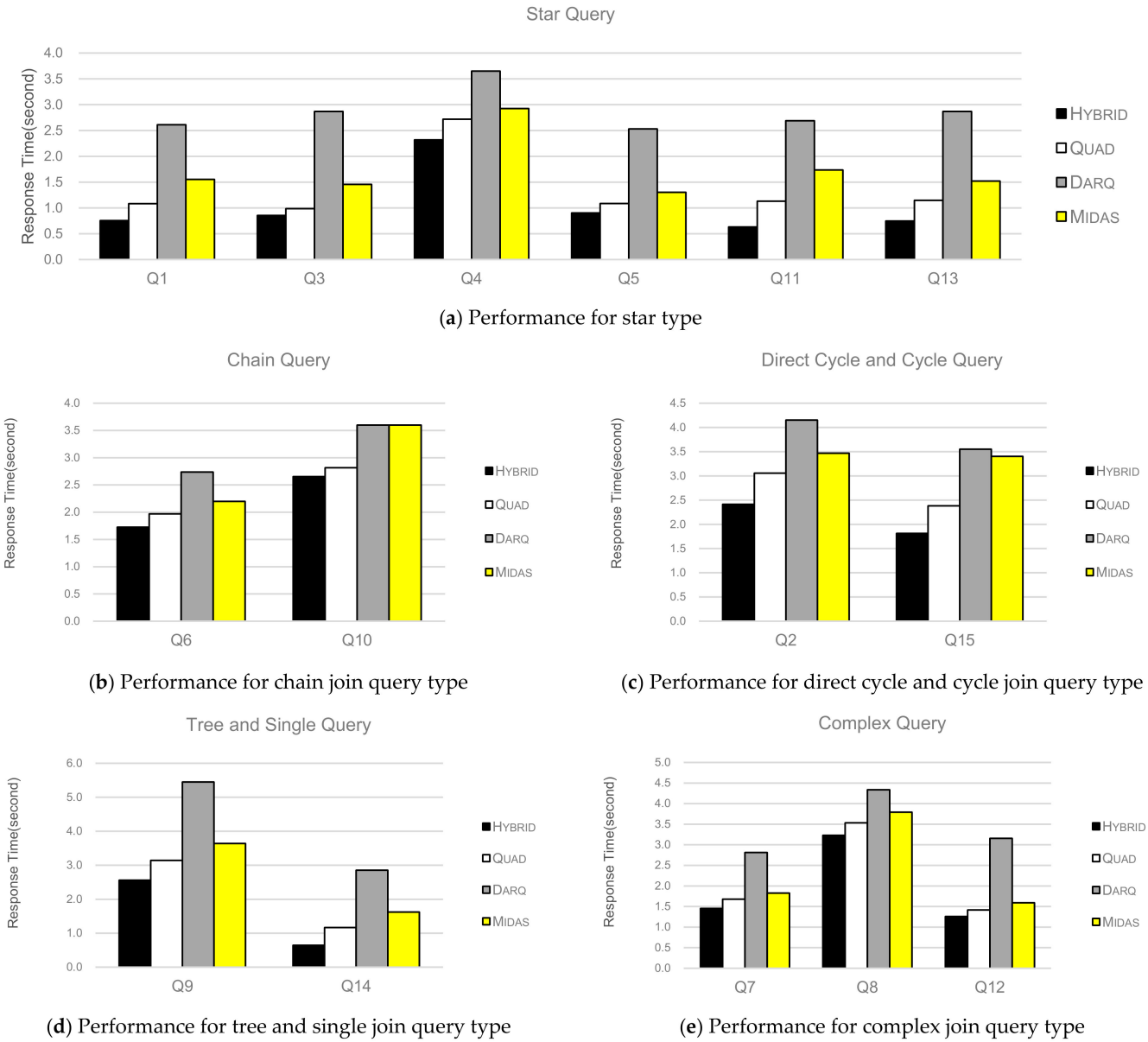
Figure 9 presents the three-dimensional points in the 3D R*-tree with respect to the LUBM dataset. All data are visualized through the interface provided by ELKI (Environment for Developing KDD-Applications Supported by Index Structures). This figure shows that all points are converted into the three-dimensional space. In the LOD cloud, the volume of data is very large, with some application areas containing a billion or more points. Such datasets usually far surpass the capabilities of a standard computing system, particularly if real-time interactions are desired. This may result in significant demand for memory and disk storage. Thus, the factors primarily influencing the system performance are the storage space and query performance.



(**a**) Three-dimensional points (**b**) 3D R*-tree

Figure 9. Three-dimensional points in the 3D R*-tree.

To evaluate the join query performance of the HYBRID method, we performed 15 SPARQL join queries that expressed the above join query types. Figure 10 illustrates the join query performance for various join query types. Experimental results show that the query performance of HYBRID is always superior to that of other methods. DARQ and MIDAS perform considerably worse than QUAD and HYBRID because these methods produce considerably larger intermediate results and require more time for I/O and CPU. QUAD shows slightly worse performance than HYBRID because QUAD needs to load six B+-trees to support all access query patterns and decompress intermediate results in a short time. However, HYBRID considers only a 3D R*-tree with *k*-d trees without storing all possible access combinations. It further reduces the intermediate results using the two-step query processing techniques described in Section 3.4.



(**a**) Performance for star type

(**b**) Performance for chain join query type

(**c**) Performance for direct cycle and cycle join query type

(**d**) Performance for tree and single join query type

(**e**) Performance for complex join query type

Figure 10. Join query performance for various join types.

The second set of experiments involves the storage amount. In Figure 11, the storage amounts are presented for HYBRID, QUAD, DARQ, and MIDAS. In terms of storage space, HYBRID is superior to QUAD for all datasets because QUAD maintains six indexes for all access patterns on the RDF triples with the context. DARQ requires the least amount of storage space because DARQ queries are processed at the SPARQL endpoints provided by remote publishers; hence, in DARQ, data need not be stored in a local server. HYBRID and MIDAS require almost the same amount of storage space. Although both attempt to save storage space by not loading all the data into a single repository, HYBRID and MIDAS perform slightly worse than DARQ because HYBRID has a 3D R*-tree with *k*-d trees, whereas MIDAS only has a *k*-d tree and requires additional space for service descriptions. QUAD performs significantly worse than the other three methods.

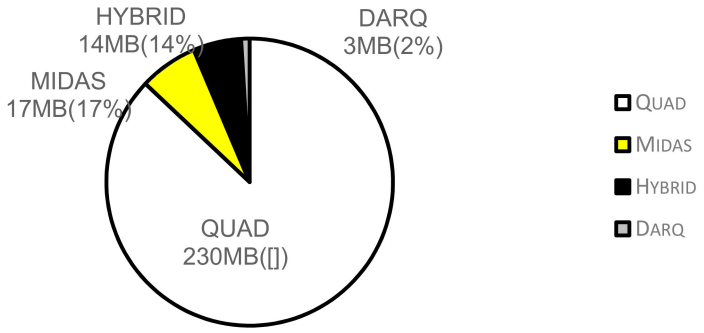


Figure 11. Storage amount.

During the join query processing, memory is allocated for retaining the filtering results. Thus, the memory size allocated for the join query processing is highly correlated with the filtering result size. Compared with QUAD, DARQ, and MIDAS, HYBRID requires the least amount of main memory because it can significantly reduce the filtering result size. Although DARQ requires the smallest storage space, it uses large memory to compute the join of intermediate results at the control site. The major motivation behind developing the HYBRID method is the desire to reduce the amount of main memory required to load the filtering results into the memory. Because the *k*-d trees are completely transferred into the main memory when the filtering results are required for the refinement phase, we assume that the main memory is sufficiently large to process these trees. If the *k*-d trees are excessively large to entirely fit in the main memory, a memory allocation problem arises. When the main memory becomes full and a new bucket must be inserted, any other bucket must be replaced. Several approaches have been proposed to determine the bucket that must be removed from the memory, including FIFO (First In First Out), LRU (Least Recently Used), and LFU (Least Frequently Used). Owing to memory space limitations, we should like to keep often-used data in the memory. Consequently, we use the LFU algorithm to manage the *k*-d trees, in which the least frequently used bucket is removed whenever the memory overflows.

The storage amount of the filtering results can be considerably large if the dataset is large. Copying such a large quantity of data into the memory is obviously time- and space-consuming. Therefore, it is essential to reduce the storage amount of the filtering results, particularly in a multiuser environment. The proposed HYBRID method can significantly decrease the amount of required memory space. Moreover, the two-step query processing achieves better CPU performance than that of the other methods. The experimental results show that the storage and index structures in HYBRID are compact and efficient; hence, only data relevant to the queries are mostly accessed. The benefits provided by the compact storage and two-step index structure lead to efficient memory utilization and reduced I/O cost.

From the above observations, we conclude that better performance is expected using HYBRID, even though it does not exhibit the best value in each criterion compared with other methods. This is because HYBRID achieves good join query performance with a small amount of storage. Therefore, we can reasonably predict that HYBRID will achieve a better overall performance than QUAD, DARQ, and MIDAS.

4.2. Performance of Hot-Cold Segment Identification Method

We measure the overall performance of our hot-cold segment identification algorithm using a currently available real LOD dataset. We used various data to obtain realistic results on the performance of large-scale RDF storage structures. To be as general as possible, this dataset was chosen from several RDF data used in the LOD cloud. Table 2 lists the characteristics of the collected dataset. To describe the characteristics of the dataset, we provide the size and the number of triples, subjects, predicates, and objects. DBpedia contains RDF information extracted from Wikipedia. DrugBank contains RDF information on

drugs. It is widely used by the drug industry, medicinal chemists, pharmacists, physicians, students, and public institutions. LinkedGeoData is a large spatial knowledge base which has been derived from OpenStreetMap for Semantic Web. These three data are widely used for developing Semantic Web applications.

Table 2. Characteristics of the Linked Open Data (LOD) dataset.

	Size (MB)	Number of Triples	Number of Subjects	Number of Predicates	Number of Objects
DBpedia	3.94	31,050	4008	23	16,644
DrugBank	144	766,920	19,693	119	274,864
LinkedGeoData	327	2,207,295	552,541	1320	1,308,214

The hot-cold segment identification algorithm determines how to allocate data between SSD and HDD. The storage system composed of a single storage medium cannot meet the demand of high capacity, low cost, and high-performance at the same time. Therefore, if multiple storage media are integrated to form a hybrid storage structure, using their respective advantages, the requirements for large capacity, low cost, and high-performance can be met. Figure 12 shows the join query performance of the real LOD dataset listed in Table 2, when the 3D R*-tree is kept in SSD and HDD, respectively. We find that the performance difference of the 3D R*-tree on SSD and HDD is not apparent through this experimental result. Hence, we put the 3D R*-tree in HDD since the cost of SSD is higher than that of the HDD. Figure 13 shows the results of relocating each datum to an appropriate storage device using hot-cold data identified by the decision tree. For different benchmark complexities, DBpedia, DrugBank, and LinkedGeoData in Table 2 are considered. The results show that an average of 42% of the SSD is used to store hot segments, and an average of 58% of the HDD is used to store cold segments. Thus, SSD is used to the minimum.

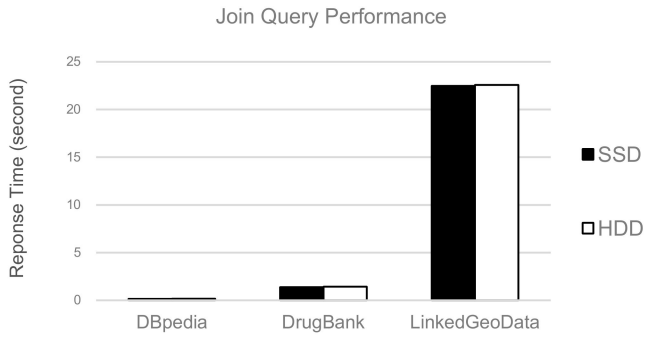


Figure 12. Three-dimensional R*-tree performance on SSD and HDD.

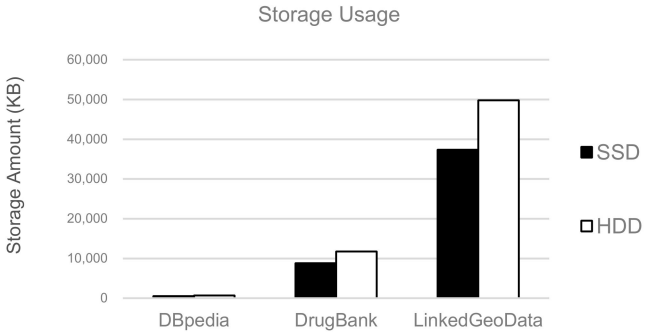


Figure 13. Results of relocating data in SSD and HDD.

Figure 14 illustrates the search performance of storage structures obtained from our experimental dataset. The experiments perform comparing response times using SSD only,

our hybrid storage structure (referred to as HSS), and HDD only, respectively. The results show that the search performance of SSD always beats the performance of other storage devices. HSS performs slightly worse than SSD because our hybrid storage structure is based on the hot-cold segment identification strategy. HDD performs significantly worse than the other storage structures. For all data, SSD performs 1.6 times faster than HDD, but it costs 2.6 times more. The performance of HSS, which balances cost and performance optimally, shows more than 70% of SSD search performance with minimal use of the expensive SSD. If the response time and cost are considered as main criteria in comparisons, we conclude that a reasonable storage structure is possible with HSS.

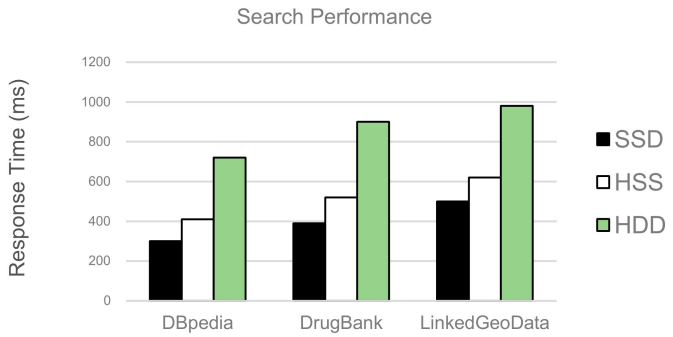


Figure 14. Search performance of storage structures.

The storage system should be able to handle read-write operations of data efficiently. The time needed for write of a certain number of RDF triples is shown in Figure 15. This figure is a graph comparing the performance of "page size write" and "immediate write" operations. The page size write means collecting data until it is a multiple of the SSD page size and then performing a write operation, and the immediate write means an immediate write operation. In this plot, we observe that the page size write operation performs better than the immediate write operation. As the number of triples increase, the difference becomes even clearer. The gap between the write time is owing to reducing unnecessary erasing operations. As a result, cell unusable states are reduced by this benefit. Even when the number of write operations is small, the performance is improved by more than 5.9 times, and as the number of write operations increases, the performance of the proposed method is more excellent.

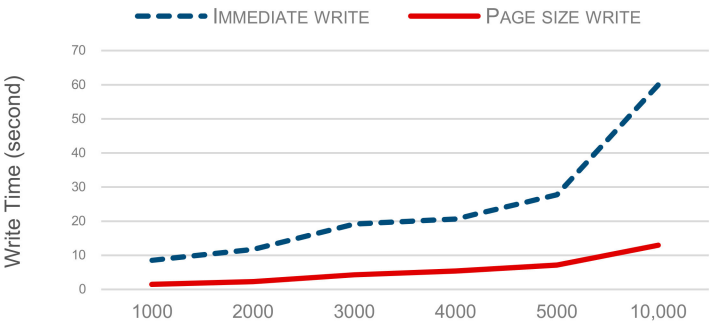


Figure 15. The time needed for write of a certain number of RDF triples.

5. Conclusions and Future Work

The efficient storage and query processing for Big RDF Data is a challenging requirement in the LOD cloud. In this paper, we proposed an extended multidimensional histogram, called MDH*, to store, compress, and discover LOD. Further, we established an MDH*-based join algorithm to increase the efficiency of MDH*. An extended RDF tuple with two special count frequencies was considered, which could help reduce join costs. We also proposed an extension of the existing index structure. Our index structure comprised two phases: one for MBB approximation using the 3D R*-tree and the other

for auxiliary main memory indexes using *k*-d trees. In contrast to existing studies on this subject, in which only the filtering phase was investigated, we considered the refinement phase that improves the main memory performance because the refinement phase exerts a critical influence on the performance of LOD query processing. The refinement phase involved time-consuming processes, particularly when accessing numerous unreliable LOD resources.

We investigated in detail a two-step index structure based on the MBB approximation and the MDH* to improve the performance of join query processing. The filtering phase quickly reduced the entire set being searched to a set of candidate MBBs using the 3D R*-tree. The refinement phase precisely evaluated the extended RDF tuples in the candidate MBBs using the *k*-d trees. The two-step index structure aims to support efficient join query processing using a hybrid storage structure and compact storage layout. We evaluated the proposed method against other popular existing methods. Experimental results indicated that our method based on the two-step index structure achieved efficient join query performance with flash memory-based *k*-d trees and a small amount of storage. In future work, it is desirable to discuss the maintenance problem of our index structure in detail. After initializing the two-step index structure, the index must be dynamically updated using up-to-date data. Recently, Vidal et al. [32] addressed this issue; however, a more detailed study is required.

Author Contributions: Writing—original draft preparation, Y.S.; Software and validation, T.Z. and S.Y.; Writing—review and editing, Y.L.; Project administration, Y.L.; Funding acquisition, Y.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2016R1D1A1B02008553).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Okoye, K. Linked Open Data: State-of-the-Art Mechanisms and Conceptual Framework. In *Linked Open Data: Applications, Trends and Future Developments*; Okoye, K., Ed.; IntechOpen: London, UK, 2020; Volume 3, pp. 158–190. [CrossRef]
2. Svoboda, M.; Mlynkova, I. Linked Data Indexing Methods: A Survey. In *On the Move to Meaningful Internet Systems: OTM 2011 Workshops, Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 7046, pp. 474–483. [CrossRef]
3. Harth, A.; Hose, K.; Karnstedt, M.; Polleres, A.; Satler, K.; Umbrich, J. Data Summaries for On-demand Queries over Linked Data. In Proceedings of the 19th International Conference on World Wide Web (WWW), Raleigh, NC, USA, 26–30 April 2010; pp. 411–420. [CrossRef]
4. Hartig, O. *Querying a Web of Linked Data: Foundations and Query Execution*; IOS Press: Amsterdam, The Netherlands, 2016; Volume 5.
5. Umbrich, J.; Hose, K.; Karnstedt, M.; Harth, A.; Polleres, A. Comparing Data Summaries for Processing Live Queries over Linked Data. *World Wide Web* **2011**, *14*, 495–544. [CrossRef]
6. Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the ACM International Conference on Management of Data (SIGMOD), New York, NY, USA, 18–21 June 1984; Volume 14, pp. 47–57. [CrossRef]
7. Umbrich, J. A Hybrid Framework for Querying Linked Data Dynamically. Ph.D. Thesis, National University of Ireland, Galway, Ireland, 2012.
8. Lynden, S.; Kojima, I.; Matono, A.; Makanura, A.; Yui, M. A Hybrid Approach to Linked Data Query Processing with Time Constraints. In Proceedings of the WWW Workshop on Linked Data on the Web (LDOW) 2013, Rio de Janeiro, Brazil, 14 May 2013.
9. Harth, A.; Decker, S. Optimized Index Structures for Querying RDF from the Web. In Proceedings of the 3rd Latin American Web Congress (LA-Web), Washington, DC, USA, 1 October–2 November 2005; pp. 71–81. [CrossRef]
10. Neumann, T.; Weikum, G. RDF-3X: A RISC-style Engine for RDF. In Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), Auckland, New Zealand, 24–30 August 2008; pp. 647–659. [CrossRef]
11. Wess, C.; Karras, P.; Bernstein, A. Hexastore: Sextuple Indexing for Semantic Web Data Management. In Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), Auckland, New Zealand, 24–30 August 2008; pp. 1008–1019. [CrossRef]

12. Atre, M.; Chaoji, V.; Zaki, M.; Hendler, J. Matrix Bit Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In Proceedings of the 19th International Conference on World Wide Web (WWW), Raleigh, NC, USA, 26–30 April 2010; pp. 41–50. [CrossRef]
13. Yuan, P.; Liu, P.; Wu, B.; Jin, H.; Zhang, W.; Liu, L. TripleBit: A Fast and Compact System for Large Scale RDF Data. In Proceedings of the 39th International Conference on Very Large Data Bases (VLDB), Riva del Garda, Trento, Italy, 30 August 2013; pp. 517–528. [CrossRef]
14. Quilitz, B.; Leser, U. Querying Distributed RDF Data Sources with SPARQL. In Proceedings of the 5th European Semantic Web Conf. (ESWC), Lecture Notes in Computer Science, Canary Islands, Spain, 27 June 2008; Volume 5021, pp. 524–538. [CrossRef]
15. Langegger, A.; Wob, W.; Blochl, M. A Semantic Middleware for Virtual Data Integration on the Web. In Proceedings of the 5th European Semantic Web Conference (ESWC), Lecture Notes in Computer Science, Canary Islands, Spain, 27 June 2008; Volume 5021, pp. 493–507. [CrossRef]
16. Abdelaziz, I.; Mansour, E.; Ouzzani, M.; Aboulnaga, A.; Kalnis, P. Lusail: A System for Querying Linked Data at Scale. In Proceedings of the 44th International Conference on Very Large Data Bases (VLDB), Rio de Janeiro, Brazil, 27–31 August 2018; pp. 485–498. [CrossRef]
17. Lyden, S.; Yui, M.; Matono, A.; Nakanura, A.; Ogawa, H.; Kojima, I. Optimising Coverage, Freshness and Diversity in Live Exploration-based Linked Data Queries. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS), Nimes, France, 13–15 June 2016; Volume 18. [CrossRef]
18. Tsatsanifos, G.; Sacharidis, D.; Sellis, T. On Enhancing Scalability for Distributed RDF/S Stores. In Proceedings of the 14th International Conference on Extending Database Technology, Uppsala, Sweden, 21–24 March 2011; pp. 141–152. [CrossRef]
19. Mountantonakis, M.; Tzitzikas, Y. Scalable Methods for Measuring the Connectivity and Quality of Large Numbers of Linked Datasets. *ACM J. Data Inf. Qual.* **2018**, *9*, 15. [CrossRef]
20. Fevgas, A.; Bozanis, P. A Spatial Index for Hybrid Storage. In Proceedings of the 23th International Database Applications & Engineering Symposium (IDEAS), Athens, Greece, 10–12 June 2019; pp. 1–8. [CrossRef]
21. Sakr, S.; Wylot, M.; Mutharaju, R.; Le Phuoc, D.; Fundulaki, I. *Linked Data: Storing, Querying, and Reasoning*; Springer: Cam, Switzerland, 2018; Volume 4, pp. 51–83. [CrossRef]
22. Linking Open Data: W3C SWEO Community Project. Available online: http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData (accessed on 12 March 2017).
23. Chawla, T.; Singh, G.; Pilli, E.S. JOTR: Join-Optimistic Triple Reordering Approach for SPARQL Query Optimization on Big RDF Data. In Proceedings of the 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Bangalore, India, 10–12 July 2018; pp. 1–7. [CrossRef]
24. Bayer, R.; McCreight, E.M. Organization and Maintenance of Large Ordered Indexes. *Acta Inform.* **1972**, *1*, 173–189. [CrossRef]
25. Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. In Proceedings of the 6th International Conference on Very Large Data Bases (VLDB), Montreal, QC, Canada, 1–3 October 1980; pp. 212–223. [CrossRef]
26. Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H.R. Extendible Hashing: A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* **1979**, *4*, 315–344. [CrossRef]
27. Moten, D. 3D R-tree in Java. Available online: https://github.com/davidmoten/rtree-3d (accessed on 13 October 2019).
28. Jin, C.; De-lin, S.; Fen-xiang, M. An Improved ID3 Decision Tree Algorithm. In Proceedings of the International Conference on Computer Science & Education (ICCSE), Nanning, China, 25–28 July 2009; pp. 127–130. [CrossRef]
29. Park, D.C.; Du, D. Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters. In Proceedings of the Mass Storage Systems and Technologies (MSST), Denver, CO, USA, 23–27 May 2011. [CrossRef]
30. Park, D.C. Hot and Cold Data Identification: Applications to Storage Devices and Systems. Ph.D. Thesis, The University of Minnesota, Minneapolis, MN, USA, 2012.
31. SWAT Projects-The Lehigh University Benchmark (LUBM). Available online: http://swat.cse.lehigh.edu/projects/lubm (accessed on 10 March 2020).
32. Vidal, V.; Casanova, M.; Menendez, E.; Arruda, N.; Pequeno, V.; Paes Leme, L. Using Changesets for Incremental Maintenance of Linkset Views. In Proceedings of the 17th International Conference on Web Information Systems Engineering (WISE), New York, NY, USA, 8–10 November 2016; pp. 196–203. [CrossRef]