SUMMARY

Graph databases have over the recent years become increasingly popular. The graph data model can be used to model real-world entities, their relationships, and domain semantics. A reason for the rapid growth of graph databases is due to their schematic flexibility and the ability to naturally model application data. Specifically, knowledge graphs (KGs) model data as a set of triples consisting of a subject, predicate, and object. KGs are stored in triplestores and queried via the SPARQL language. A SPARQL query in its most basic form consists of one or more triple patterns which describe subgraph patterns to match against. Evaluating two overlapping triple patterns requires joined over the overlapping variable(s). These joins can be expensive to compute on disk-based triplestores due to expensive disk operations. Join operations on triple patterns containing a single variable on which the join is performed will perform existence checks. These existence checks will in some triplestores perform disk lookup which is expensive and unnecessary. Specifically, Apache Jena is one such triplestore.

We conduct an extensive study of the Apache Jena query engine and propose to integrate a Bloom filter (BF) to skip disk-based lookups when a given triple pattern does not exist. We call our approach *JenaBloom*. A BF is a compact bit vector and a set of hash functions used to flip bits in the bit vector. An item is hashed using all of the BF hash functions, and the corresponding bit vector bits are flipped to 1s. During existence checks, the same hash functions are applied to check whether all of the corresponding bits have been flipped to 1s. We use BF existence check for triple patterns only containing the one variable being joined on. When the BF returns negative, i.e., the triple pattern does not exist, the standard disk-based indexes are skipped. This saves a significant amount of time. However, BF positives are not guaranteed to be true. Hence, it is required to also perform existence check in the standard disk-based indexes to verify the triple pattern existence. Therefore, both BF true and false positives induce additional runtime, as the BF existence check becomes an additional step to the standard existence check using the standard indexes. It is therefore essential to apply BF existence only for triple patterns involved in joins that produce enough BF negatives such that BF existence check optimizes query execution. We therefore experiment with the application of triple predicate statistics and apply BF existence check only for triple patterns containing predicates that belong to the top 80th percentile of most frequent predicates in the KG. The aim with this statistical approach is use BF existence only triple patterns that have larger result sets which are likely to perform a higher number of existence checks.

We create a benchmark consisting of queries used to evaluate a worst-case optimal join (WCOJ) approach in Leapfrog Triejoin and queries used to evaluate MilleniumDB. From these queries, we generate a set of queries that return empty results set. We furthermore manually handcraft queries that are designed to maximize existence checks. Our experiments clearly show that we are able to improve query execution time for queries that return empty result sets. These queries time out in Apache Jena but terminate successfully in JenaBloom. However, JenaBloom is not able to outperform Apache Jena on the remaining queries. Leapfrog Triejoin also proves to outperform Apache Jena and JenaBloom for almost any type of query. The reason JenaBloom does not outperform Apache Jena for some queries is because the fraction of BF negatives is too small. Thus, the sum of additional runtime gained from the positive BF existence checks out-balances the runtime saved from BF negatives. Finally, it is evident that JenaBloom applying statistics to predict when to apply BF existence check does not perform this prediction well, and hence, this approach does once again not improve runtime. This is due to the statistical approach being too simplistic and not considering that a frequent triple predicate is also more likely to produce BF positives.

Jena Bloom: Query Execution with Bloom Filter Existence Check

Martin Pekár Christensen
Aalborg University
Denmark
mpch@cs.aau.dk

Matteo Lissandrini
Aalborg University
Denmark
matteo@cs.aau.dk

Katja Hose*
TU Wien
Austria
katja.hose@tuwien.ac.at

ABSTRACT

In recent years, graph data management, triplestores, and knowledge graphs have increasingly have attracted interest. However, it still remains challenging to efficiently query triplestores, as many optimization strategies from traditional databases are still left unexplored. As a first step to optimize triplestores, this paper examines the question of how to improve query execution time by addressing costly existence checks in join operations. To achieve this goal, we integrate a Bloom filter residing entirely and compactly in-memory to be used in place of disk-based indexes for existence check operations. We furthermore apply triple statistics in determining the specific join operations in which Bloom filter existence checks benefit execution time. We extend a reference triplestore (Jena) with Bloom Filters and integrate our approach for query optimization.

We evaluate our approach, JenaBloom, on a large set of more than 1,500 queries, and show its effectiveness on queries returning empty result sets, as well as those returning non-empty result sets.

1 INTRODUCTION

In recent years, graph DBMS have arisen as a specialized type of DBMS to store and analyze data modelled via the graph data model [2, 13, 22]. Recently, there has been an increasing interest in graph databases [2], where the graph data model is employed to model real-world entities and their intermediate relationships, as well as the desired domain semantics. Typically, graph databases are employed where data inter-connectivity or topology are important, as well as schematic flexibility [1, 2]. Hence, many applications where inter-connectivity is key have moved from traditional database models, such as relational databases and object-oriented databases, to graph databases. One of the advantages of graph databases is that they allow to model application data more naturally. This allows graph database queries to refer directly to the graph structure, as well as applying graph-oriented operations and constraints over the graph structure. It furthermore allows for executing common graph algorithms, such as finding shortest path and finding sub-graphs.

One particular graph data model that has gained interest recently is that of Knowledge Graphs (KGs) [7, 17], i.e., databases that model information as a set of subject-predicate-object (s, p, o) triples, also called facts or statements. An (s, p, o) triple represents entities s and o connected by a predicate p representing the relation between the two entities, e.g., $\langle Michael\_Jordan,\ birthplace,\ Brooklyn \rangle$ (see Figure 1).

KGs are typically stored in graph database management systems (GDBMSs), referred to as triplestores [22] and queried via the SPARQL query language [19] (see example query in Listing 1).

Typical graph queries require to find all the matches of substructures (i.e., pattern matching of subgraphs) within the larger graph, these are called triple patterns and basic graph patterns (BGPs). These operations are usually modelled in logical plans as a series of joins on relations corresponding to the edge types. Their query optimization is challenging because of the hardness of problems like query cardinality prediction [18], which can lead to highly under-performing execution plans.

```
1 PREFIX db: <http://dbpedia.org/resource/>
2 PREFIX dbp: <http://dbpedia.org/property/>
3 SELECT ?o WHERE {
4     db:iri5 dbp:team ?o .
5     ?o rdf:type db:iri7
6 }
```

Listing 1: Wikidata SPARQL query.

For example, some queries that require joins between two triple pattern will perform existence checks. As an example, consider the query in Listing 1. This query asks for the team of `iri5` which is of type `iri7`. The query engine will solve the first triple pattern to find intermediate solutions to variable `?o`. It will then substitute these solutions into the same variable in the second triple pattern and check for existence of this second triple pattern. Therefore, many existence check operations will be performed: one for each intermediate solution of the variable `?o` from the first triple pattern. These existence check operations will perform disk operations in disk-based triplestores which are expensive in terms of runtime. Moreover, if a triple pattern with a large result set is joined with another triple pattern, and the intersection between the two is small, many existence checks will be performed, most of which return negative using standard indexes that do not suffer from false positives. Therefore, a Bloom filter (BF) can be utilized to remove the runtime overhead of disk-based existence checks for triple patterns that do not exist.

Therefore, in this work, we propose JenaBloom: BF existence check for triple pattern joins. In practice, we analyze a reference open-source triplestore implementation, Apache Jena [1], which executes queries as a series of index-nested loop joins. Then, we extend its query execution via the use of BFs, a typical optimization to reduce disk-based index access which was not implemented in this and similar systems [22]. Therefore, we implement a BF for triples to substitute the cost of disk operations when the join operation requires to check if a given triple exists in the database. We call our proposal *JenaBloom* and implement it in Apache Jena 3.17. Our evaluation (Section 5) shows that queries with empty result sets time out in Jena but are executed efficiently in JenaBloom. We show in our experiments the type of queries and constraints required for

*Also with Aalborg University, khose@cs.aau.dk.

[1] https://jena.apache.org/

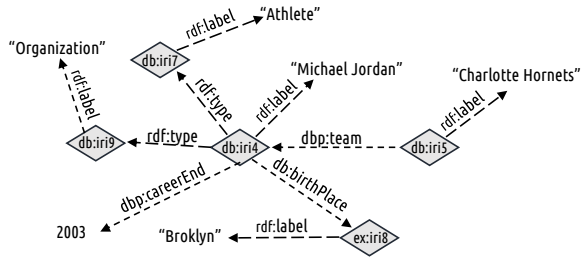

Figure 1: Sample KG describing Michael Jordan.

BFs to be efficient in query execution, as well as the case in which BFs are inefficient. Since the use of BFs benefits only particular queries with high degree of negatives, queries with high positive rates on joins could actually experience a slowdown. Therefore, we experiment with the application of statistics over stored triples to determine when to apply BF existence check. Therefore, our contributions are the following:

(1) We study the query engine of Apache Jena and propose an integration strategy of BF existence checks.
(2) We implement a prototype applying BFs in the query execution module of Apache Jena, showing the optimization opportunity of this technique in SPARQL query execution, identifying advantages and limitations.
(3) We implement the application of triple statistics in determining when to apply BF existence checks and evaluate its effect on queries in which BF existence checks are inefficient.

The rest of this paper is structured as follows: Section 2 presents necessary preliminaries of RDF triplestores and BFs, Section 3 presents related work, and Section 4 presents the methodology behind query execution using BF existence check. Finally, we evaluate our approach in Section 5 and conclude the results in Section 6 as well as present some future directions for this research problem.

2 PROBLEM DEFINITION

In this section, we introduce the definition of the RDF data model behind KGs and the SPARQL query language. We further introduce how Bloom filters can be used for SPARQL query execution, and thus provide a problem definition.

2.1 RDF and SPARQL

Knowledge graphs are usually stored using the data model provided by the Resource Description Framework (RDF) [12]. RDF KGs are represented as a directed graph consisting of *triple*s (see Figure 1).

Definition 2.1 (Knowledge Graph). An RDF knowledge graph is a set of triples where a triple $t{:}(s,p,o)$ consists of a subject s, predicate p, and an object o, where s and o are KG nodes connected by an edge with label p. Thus, given, the set of IRIs (identifiers of entities) I, the set of blank nodes (placeholder for nodes) B, and the set of literals (named constants) L, we have that $s{\in}I{\cup}B$, $p{\in}I$ and $o{\in}I{\cup}B{\cup}L$, which means a triple $t{:}(s,p,o){\in}(I{\cup}B){\times}I{\times}(I{\cup}B{\cup}L)$.

SPARQL [26] is the standard query language to query RDF data. In its simplest form, a SPARQL query contains a projection clause and a *basic graph pattern* (BGP) of triple patterns [16]. A triple pattern consists of a *subject*, *predicate*, and *object*, following the definition of a triple, but with the addition of a variable $?x{\in}\mathcal{X}$ from a set of named variables distinct from I, B and L, i.e., $\mathcal{X}\cap(I{\cup}B{\cup}L)=\emptyset$. The goal of a query engine when executing a SPARQL query is, among other things, to compute bindings for the triple pattern variables and execute joins between triple patterns, such that, when replacing the variables with the bindings, the triples obtained in this way exists in the KG.

For example, consider the SPARQL query in Listing 1. The query asks for all objects `o` that appears in triples where `s` is `iri5` and `p` is `db:team`, as well as in triples where `p` is `rdf:type` and `o` is `db:iri7`. [2] Thus, here we have two triple patterns joined on the single variable `?o` with solution `db:iri4` in Figure 1.

Given a BGP, the responsibility of the query engine is to iterate triple patterns contained within the BGP to compute variable solutions. Variable solutions are used in join operations between triple patterns as mentioned earlier and to filter solutions using existence checks. This results in a final set of variable solutions which become the BGP output.

2.2 Query optimization with Bloom Filters

A BF [4] is a data structure which is used for fast membership filtering in constant time complexity and has the advantage of being very compact in size. A BF is an array of bits, initially all set to 0, and a collection of hash functions used to check for existence or to flip 0-bits when inserting items. Whenever an element is to be inserted into the set, a specific number of hash functions are applied on the element to determine which bits in the array must be set to 1. To check for existence of an element, the same set of hash functions are applied to check whether all bits in the array positions that the hash functions point to are set to 1. If not, the element is guaranteed to not exist in the set. However, even in case all the required bits are set to 1, the element might still not exist in the set because another combination of inserted object might have flipped the exact same set of bit array positions. Hence, BFs suffer from false positives. The false positive rate is tunable at the expense of consuming more memory, i.e., the false positive rate can be decreased by increasing the size of the bit array, as well as the number of hash functions. The probability of false positives [15] for a BF of m bit array bits, n inserted elements, and k hash functions can be described as

$$\epsilon = (1-(1-\frac{1}{m})^{kn})^k \tag{1}$$

The optimal number of bits in the BF can be approximated as

$$m = -\frac{n*ln(\epsilon)}{ln(2)^2} \tag{2}$$

Hence, as an example, given 1,000 elements to be inserted into the BF, a false positive rate of 0.01, and 5 hash functions as parameters, the size of the bit array is approximately $-1,000*ln(0.01)/ln(2)^2 = 9,585$ bits. The actual probability of false positives is then $(1-(1-1/9,585)^{5*1,000})^5 = 0.011$.

Given their *one-sided error*, BFs have often been used to reduce data-access in many application algorithms [11, 24], and their application within index-nested-loop-join algorithms is a classical

[2] Here, every non variable atom is an IRI where `db:` and `dbp:` are shorthand prefixes.

textbook example where the BF is probed in constant time before accessing a B$^+$-tree (which requires logarithmic time).

A prototypical implementation in a triplestore, e.g., in Apache Jena, is to store triples in a relation and store it within a B$^+$-tree. In practice, Apache Jena deploys a set of at least 3 B$^+$-trees to store in different orders the elements of each triple, e.g., a tree for SPO, a tree for OPS, and a tree for PSO. Therefore, to find bindings for a triple pattern, e.g., the first one in Listing 1, one would first access one of the trees, e.g., the SPO three to find all ?*o* for the prefix (`db:ir5`, `dbp:team`), and then for each value found in this way, another index would be accessed to identify values that satisfy the *join* condition, e.g., the SPO again for each value for ?*o*. Thus, in the second step, accessing the index equates to an existence check, i.e., given that we found the triple (`db:ir5`, `dbp:team`, `db:iri4`), verify if the triple (`db:ir4`, `rdf:type`, `db:iri7`) exists in the graph. For very large graphs, in the case where the first triple pattern produces a large number of results, the application of BFs when solving the second triple pattern could then save a huge amount of index accesses. Yet, in the case where most of the results of the first triple pattern lead to triples that actually exist in the graph, accessing the BF will instead cause a decrease in performance. Furthermore, BFs introduce false positives for triples that do not exist. Hence, when the BF returns positive, the B$^+$-tree needs to be accessed to verify the BF positive. This means that the BF existence check becomes an additional step in the existence check operation and thereby introduces runtime overhead for BF positives.

2.3 Problem Definition

In this work, we aim at removing expensive disk-based index lookups on join operations by employing BFs in place of standard indexes on existence checks. We formally define our problem as follows:

Problem 1 (Query Processing Existence Check). *Given a knowledge graph G and a query Q, minimize the number of disk operations.*

In this work, joins are performed using nested-index-loop join over B$^+$-trees implemented by default in Apache Jena. In JenaBloom, the joins also adopt BF existence checks.

Therefore, we perform BF existence check when joining two triple patterns, and when the triple pattern being evaluated contains no other variables than the variable included in the join operation. On BF negatives, the index existence check is skipped. However, on positives, index existence check is still required, due to the probability of false positives. This can drastically under-perform when compared to the standard one. Therefore, we define the following problem and experiment with applying statistics in determining when to apply BF existence check given the predicate of the triple pattern.

Problem 2 (BF Existence Check Under-Performance). *Given a query Q and the optimized and the corresponding unoptimized query processing plans O and U, respectively, determine which of O and U is most efficient for each triple pattern $tp_i \in Q$.*

3 RELATED WORK

We present here the different indexing approaches as well as query optimization techniques for improving query evaluation scalability.

3.1 Triplestore Indexing

Hexastore [27] and RDF-3X [16] are RDF triplestores proposing to optimize query performance by sacrificing index space, where 6 distinct B$^+$-tree indexes are used to materialize all possible combinations of triple terms. This allows for using an index that is optimal for a given triple pattern, whereas Apache Jena only uses 3 distinct indexes. Leapfrog Triejoin [8] is a worst-case optimal join implementation in Apache Jena and is another example that adds three additional B$^+$-tree indexes to a total of 6 indexes in order to implement this join. Hence, these approaches introduce index redundancy, as defined by T. Sagi et al [22] Similarly, we perform BF existence check on a subset of join operations: those that contain only the single variable that the triple patterns are being joined on. This additionally requires a populated BF which in turn also introduces a low level of redundancy due to the BF compactness.

Alternatively, other RDF triplestores, including Virtuoso [6] and BitMat [3], use bit maps to optimize the performance of join queries. A bit map is simply a bit sequence, and a bit map index on an attribute consists of one bit map for each value the attribute can take [23]. The number of bit map bits is determined by the number of entities in the KG. The ith bit of the bit map is set to 1 if the entity number i has the value the bit map represents for the attribute corresponding to the bit map. As an example, consider entity `db:iri4` in Figure 1 and its property `rdf:type`, and let the universe of distinct types be T. This means $|T|$ bit vectors are needed, one for each type type $t \in T$. The dimension of the bit vectors are equal to the number of entities in the KG. As entity `db:iri4` has type `db:iri7`, the bit vector representing the type `db:iri7` has the bit in the position corresponding to entity `db:iri4` flipped to a 1.

Bit maps can also be used for existence check in constant time complexity, but are not as compact compared to BFs due to the large number of bit vectors for all possible distinct values for all distinct entity properties. BitMat is an example of a triplestore that utilizes bit maps for existence checks.

3.2 Optimized Indexes

Hash indexes are popular index choices for point lookup queries. Both MH-Index [14] and Redis++ [28] propose dynamic hashing schemes that optimize query efficiency with a two-level hash index. The MH-Index is based on the linked list structure, where each element in the linked list is referred to as a *cell*. However, unlike traditional linked lists, cells within the MH-Index can be retrieved in constant time by hashing. Hence, the MH-Index does not suffer from the linear time complexity when retrieving cells as in traditional linked lists. As in dynamic hashing schemes, each MH-Index cell or Redis++ bucket consists of multiple nodes, where each node contains the data records to be inserted into the index. However, these indexes are only applicable for point lookup queries and are therefore not applicable in the evaluation of triple pattern range queries. Second, the indexes are not compact and would not be a scalable solution when applying them only for existence checking.

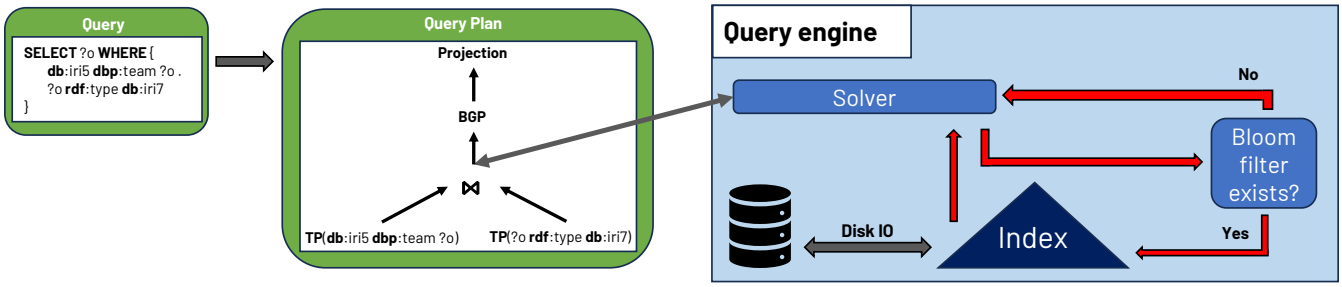

Figure 2: Approach architecture.

Learned indexes have become increasingly popular in database systems as they prove to improve indexing performance over the traditional data structures, such as the B$^+$-tree [5, 10]. However, learned indexes are designed to improve the general index performance and are therefore not well-suited for existence checking alone due to the indexes still requiring disk operations.

3.3 Existence Check

FBF [11] is an enhanced BF that enhances capabilities of Bloom Filters by supporting fuzzy membership queries. Given a set of strings, the goal is to compute whether a query string exists in the set by computing a score for the query string. This score is dependent on the degree of similarity to the set of strings. When inserting strings, the input string is hashed using locality-sensitive hashing, where similar strings get similar hashes. Thus, strings with similar hashes are inserted into the same bin. There are already many works that use locality-sensitive hashing in BFs [9, 20, 21]. When querying for the existence of a string, the query string is hashed the same way using locality-sensitive hashing, and a bin is chosen based on this hash. Existence of a query string is then computed by the similarity between the query string and the strings in the bin. However, the FBF introduces false negatives which makes it unsuitable for existence checking in triplestores, as it cannot guarantee that a triple pattern does not exist. Therefore, an FBF may incorrectly skip index lookup.

4 EXISTENCE CHECK

The most common index data structure employed in triplestores is the B$^+$-tree. During query processing, the B$^+$-tree is utilized to find the mappings to triple pattern variables. These mappings are in practice disk locations in which the variable solutions can be found. These solutions form the intermediate result set of the triple pattern, and the intermediate result set will be combined with other triple patterns using a component we refer to as the *solver*. Specifically, given the intermediate result set of one triple pattern, the solver will join the triple pattern with other triple patterns using the intermediate results. After performing the triple pattern joins, the intermediate result set will be passed to the individual query operators in the classical tree-structured fashion.

The architecture of JenaBloom with a Bloom filter is depicted in Figure 2. We have chosen to integrate BF from Google's Guava 31.1 library into Apache Jena TDB version 3.17. This BF hashes elements into a given numeric datatype. The BF in JenaBloom is loaded with all stored KG triples by inserting each triple on by one when loading the triplestore. That is, every triple is hashed using a single hash function, and the corresponding bit in the BF bit array is flipped to a 1. We initially discovered that using 4-byte integers led to many triple hash collisions due to the fact that 4-byte integers can barely represent the number of stored triples in our experiments. We therefore opted to use 8-byte integer values which results in a more uniform hash distribution. We use a modular hashing function in our BF, as we observed this function to perform well compared to other hash functions.

The evaluation of some queries include performing existence checks. Disk operations performed from existence checks are often redundant, as existence checks seek Boolean answers and can to some degree be answered fully in-memory since it is only of interest knowing whether a given index key exists. Hence, a significant amount of runtime can potentially be saved by minimizing the amount of disk operations performed during query execution.

As an example, consider the SPARQL query in Listing 1 of two triple patterns joined on the single variable ?*o*. In case the triple pattern on line 4 has a large result set S for variable ?*o*, the second triple pattern on line 5 will be evaluated by $|S|$ existence checks to filter away those solutions for ?*o* in the first triple pattern that are not solutions to ?*o* in the second triple pattern. This leads to a large number of IO operations performed by index lookups. The size of the intersection of variable ?*o* shared in the two triple patterns is thus the number of positive existence checks.

4.1 Bloom Filter Existence Check

As mentioned earlier, BF existence checks are only applied on triple patterns containing no other variables than the one that the join is performed on. As seen in Figure 2, a query is given as input to the query engine. The central component of the query engine is the *Solver* which is responsible for computing variable bindings and joining triple patterns. For each triple pattern suited for existence check, the solver first performs existence check using the BF before using one of the standard indexes.

We implement two options for the application of BFs during BGP evaluation in Apache Jena: (1) where BF existence check is applied whenever a given triple pattern in the BGP is fully concrete and (2) when our statistical approach determines that the triple pattern is frequent. The fist option pays the penalty of adding runtime on positive existence check, as this would require performing B$^+$-tree existence checks to guarantee true positiveness. The second

option tries to mitigate this penalty by not using BF existence check during the evaluation of a given triple pattern when evaluating triple pattern containing infrequent predicates. The second option will be described in the following section.

During query execution, Jena performs a series of triple pattern evaluations that require B$^+$-tree lookups. Hence, the theoretical query execution runtime for Jena is defined as

$$Jena_R = I * rt(BT) \tag{3}$$

where I is the number of (B$^+$-tree) index searches and rt is a function returning the runtime of a single B$^+$-tree (BT) search. Then, using BF existence check, a fraction of the B$^+$-tree index searches are avoided due to BF negatives, denoted BF_n. BF positives, denoted BF_p, adds additional runtime because B$^+$-tree search is then required. We define the theoretical runtime of JenaBloom for index searches that cannot use BF existence check as $JenaB_{R_I} = (I - (BF_p + BF_n)) * rt(BT)$, the runtime when using BF existence check that return positives as $JenaB_{R_{BF_p}} = BF_p * (rt(BT) + rt(BF))$, and the runtime when BF existence checks return negative as $JenaB_{R_{BF_n}} = BF_n * runtime(BF)$. Aggregating these runtimes, we define the final, theoretical runtime for JenaBloom as

$$JenaB_R = JenaB_{R_I} + JenaB_{R_{BF_p}} + JenaB_{R_{BF_n}} \tag{4}$$

Using Equations 3 and 4, we derive the following constraint to be satisfied for JenaBloom to be faster than Jena:

$$Jena_R \geq BF_p * rt(BF) + I * rt(BT) - BF_n * rt(BT) + BF_n * rt(BF) \tag{5}$$

Based on experiments, $rt(BT) = 2771$ns and $rt(BF) = 734$ns.

4.2 Existence Check Using Statistics

So far, BF existence check is applied on every evaluation of a triple pattern containing no other variables than those being joined on. This can be a disadvantage for some queries that result in many BF positives, as mentioned in Section 2.2. For this reason, we experiment with the application of collected statistics about stored triples to determine whether to use BF existence check for a given triple pattern. Specifically, we collect the frequency of predicates among the stored triples and save those predicates that belong to the top 80th percentile. Triple patterns with frequent predicates are more likely to have large result sets. Hence, evaluating these triple patterns requires performing larger joins with more BF existence checks. Therefore, for a given triple pattern, we first check whether the frequency of the predicate of the triple pattern belongs to the 80th predicate frequency percentile. If so, BF existence check will be applied.

5 EVALUATION

We evaluate the runtime performance when applying JenaBloom in triplestores to answer whether BF existence checking is effective in removing redundant disk operations. We furthermore evaluate whether using statistics is effective in predicting when to apply BF existence check to minimize the runtime overhead introduced by BF false positives. Specifically, we apply our BF existence check approach and evaluate the runtime improvement against the triplestore *Jena* and Leapfrog Triejoin [8]. We refer to the baselines as *Jena* and *Leapfrog*, respectively, for the remainder of this section. We query JenaBloom and the baselines using Fuseki 2.

Our code base[3] and benchmark[4] are available on GitHub.

We run the experiments on a server with 2TB of RAM and a 64-core CPU.

5.1 Benchmark

Original

```
SELECT (COUNT(*) AS ?count) WHERE {
    ?x1 <http://www.wikidata.org/prop/direct/P485> ?x2 .
    ?x2 <http://www.wikidata.org/prop/direct/P4195> ?x3 .
    ?x3 <http://www.wikidata.org/prop/direct/P1753> ?x4
}
```

Empty

```
SELECT (COUNT(*) AS ?count) WHERE {
    ?x1 <http://www.wikidata.org/prop/direct/P136> ?x2 .
    ?x2 <http://www.wikidata.org/prop/direct/P4195> ?x3 .
    ?x3 <http://www.wikidata.org/prop/direct/P1753> ?x4
}
```

Modified

```
SELECT (COUNT(*) AS ?count) WHERE {
    <http://www.wikidata.org/entity/Q43878362> ?p ?x2 .
    ?x2 <http://www.wikidata.org/prop/direct/P4195> ?x3 .
    ?x3 <http://www.wikidata.org/prop/direct/P1753> ?x4
}
```

Figure 3: Generation of benchmark queries.

Table 1: Number of queries in each query group, number of queries slower than 100ms in Jena, and their description.

Type	# queries	# slow	Description
BGP	850	526	Leapfrog experiment queries containing simple BGPs
MilleniumDB	436	323	MilleniumDB experiment queries
Empty	1286	540	Return empty result set
Modified	474	67	Return small result set
Existence check	119	71	Specialized to maximize negative existence checks in joins

We evaluate on the Wikidata KG consisting of 1,838,908,292 triples. Our benchmark consists of 3,315 queries of different shapes based on the benchmark used in the evaluation of Leapfrog Triejoin[5]. We group our set of queries depending on their characteristics, summarized in Table 1. These groups of queries are further grouped into sub-groups, depending on shape. We have also included the

[3] https://github.com/dkw-aau/jenaclone-3.17
[4] https://github.com/dkw-aau/leapfrog-rdf-benchmark
[5] https://github.com/GQgH5wFgzT/benchmark-leapfrog

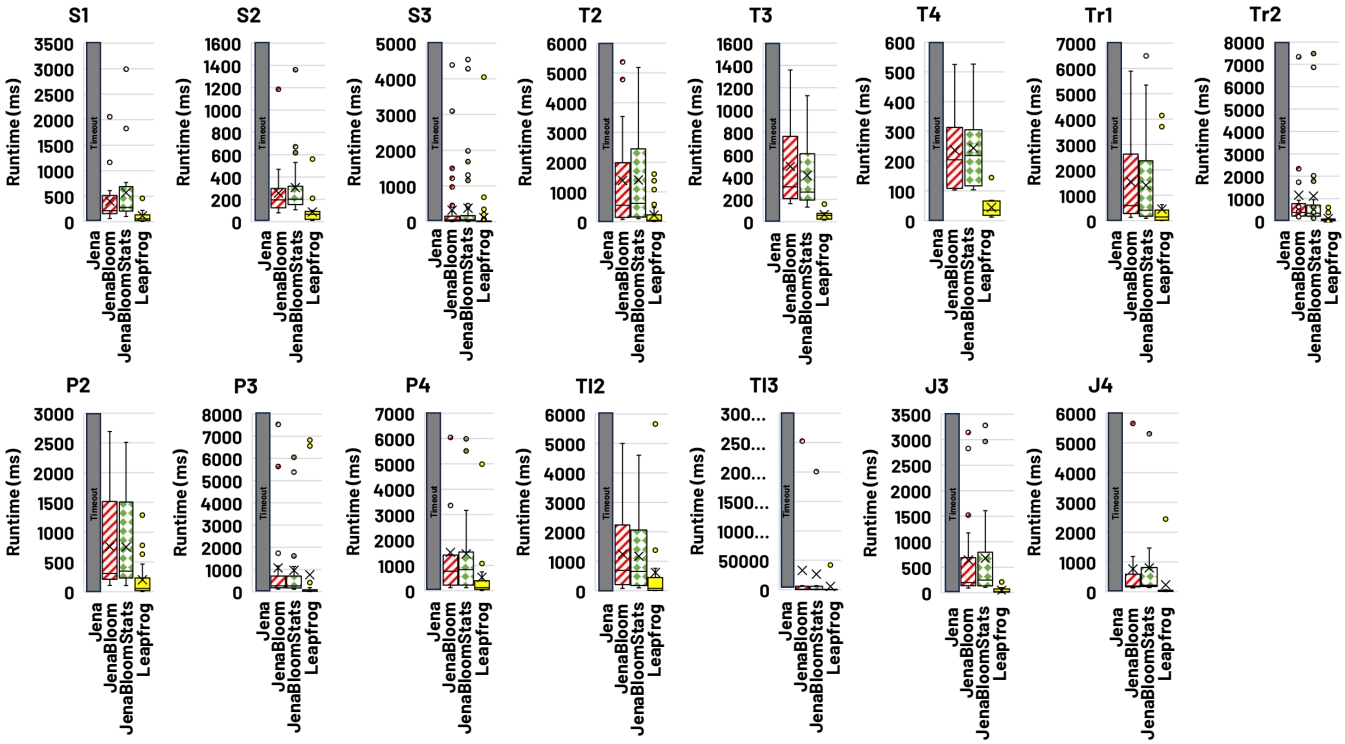

Figure 4: Performance on *Empty* queries.

set of queries used in the evaluation of MilleniumDB [25]. Finally, we filter our full set of queries such that we keep those that take at least 100ms to execute in Jena. This leaves us with 1,615 queries which also is reflected in Table 1.

The *Empty* queries have been generated from the *BGP* queries, where a random query predicate is substituted with a predicate that belongs to the top-100 most frequent predicates, excluding *rdf:Type* and *rdf:Label*. These queries return an empty result set. The *Modified* queries are also generated from the *BGP* queries, but where the first triple pattern predicate is a variable instead of a URI, and every occurrence of the subject variable in the first triple pattern is substituted with the same variable solution from the original *BGP* query. Finally, the *Existence check* queries are manually designed such that they perform a high number of existence checks, and the intersections between the joined triple patterns are minimal. Figure 3 shows the original query and how it has been modified according to the different query groups (except for existence check queries).

We measure the average runtimes as the response time from issuing the query from the client to receiving an output. We measured the average network delay for a subset of queries to be 6.65 ms.

5.2 Bloom Filter Existence Check

We have experimented with different set of parameters to choose the optimal ones for our experiments. We use a single, modular hash function and a false positive rate of 1%. The size of the BF bit array is adjusted to fulfill these parameters.

We observe that JenaBloom outperforms Jena for queries with empty result sets (Figure 4). All of these queries perform existence checks which most often are negative, as the result sets are empty. Hence, JenaBloom outperforms Jena which times out for all of these queries.

However, JenaBloom did not outperform Jena on the queries designed specifically for existence check (Figure 5). These queries are mostly cyclic and have small result sets. Specifically, these queries contain joins with large, almost disjoint, intermediate result sets. When joining triple patterns, Jena computes solutions to the variables of the first triple patterns and inserts these solutions into the the overlapping variables of the second triple pattern. For each variable solution, Jena performs an existence check of the second triple pattern using the solutions and filters out solutions that return negative. When the BF is used for existence check and returns negative, runtime is saved, as the B^+-tree lookups are omitted. However, because some of the existence checks return positive (some are false positives), the BF existence check adds additional runtime, and hence, JenaBloom does not outperform Jena in Figure 5. The fraction of negatives among all existence checks required for JenaBloom to be faster is shown in Figure 7. The figure shows that the queries that are faster in JenaBloom perform a higher fraction of BF existence checks that are negative. Specifically, queries faster in JenaBloom than Jena in our benchmark perform on average 3.3 times more negative BF existence checks than those queries that

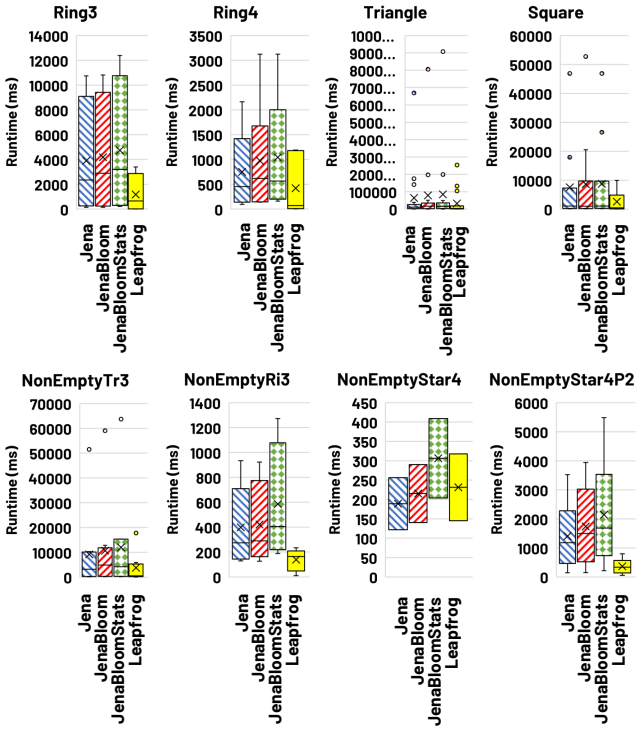

Figure 5: Performance on *Existence* queries.

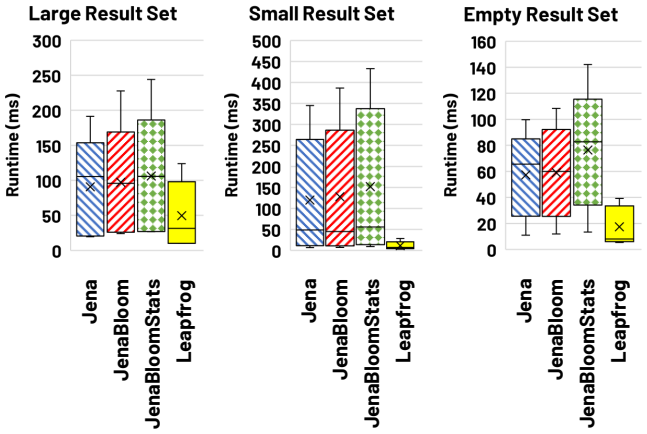

Figure 6: Runtime for large (> 350), small (< 50), and empty result set sizes.

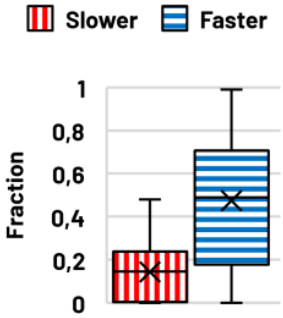

Figure 7: Fraction of existence checks with negative answers.

are slower. Hence, the fraction of positive BF existence checks is too high in most of the queries plotted in Figures 5, 8, and 9.

Finally, it is clear that Leapfrog is several times faster than any of the other approaches. However, there are exactly 112/1,615 queries that are faster in JenaBloom than in Leapfrog. Specifically, Listings 2, 3, and 4 are examples of queries where JenaBloom is faster than Leapfrog. For the query in Listing 2 from the *BGP* query group, JenaBloom improves runtime over Leapfrog by 11.6%. For the query in Listing 3 from the *Existence check* query group, the improvement is 39.1%. Finally, the query in Listing 4 is from the *Modified* query group and has an improvement of 10.2%. These three queries are examples of queries where the fraction of negative BF existence checks is great enough such that more runtimes is saved than gained from positive BF existence checks.

5.3 Bloom Filter Existence Check with Statistics

We described in Section 4.2 our approach using predicate statistics to determine when to apply BF existence checks. We observe that JenaBloom with statistics does not outperform JenaBloom for the majority of queries, and there are only improvements on a few queries among the *Empty* queries. Specifically, JenaBloom using statistics for the query shapes among the *Empty* queries in Figure 4 labelled *T3*, *Tr1*, *Tr2*, and *TI2* are faster than JenaBloom. Among the *BGP* queries in Figure 8, JenaBloom using statistics improves the runtime of JenaBloom for query shapes labelled *T2*, *Tr1*, and *Tr2*. Therefore, this approach is generally not performing well in identifying when to apply BF existence checks during query execution. This is due to our approach using statistics is naive and does not consider other triple patterns in its decision on when to apply BF existence checks. For example, consider the query in Listing 1 containing two triple patterns joined on the variable ?*o*. BF existence check will be enabled because the predicate `rdf:type` in the triple pattern that perform existence checks is frequent. Hence, the second triple pattern is more likely to perform positive BF existence checks compared a triple pattern containing an infrequent predicate. On the other hand, if we substitute the frequent predicate `rdf:type` with an infrequent predicate, BF existence check will be disabled, and the runtime of this approach will be the same as Jena. Therefore, it is unlikely that JenaBloom using statistics will improve the runtime of JenaBloom, as it is likely that queries where BF existence check is enabled using statistics result in too many BF existence check positives.

```
1 PREFIX wdp: <http://www.wikidata.org/prop/direct/>
2 SELECT (COUNT(*) AS ?count) WHERE {
3     ?x1 wdp:P5130 ?x2 .
4     ?x2 wdp:P463 ?x3
5 }
```

Listing 2: Wikidata SPARQL query.

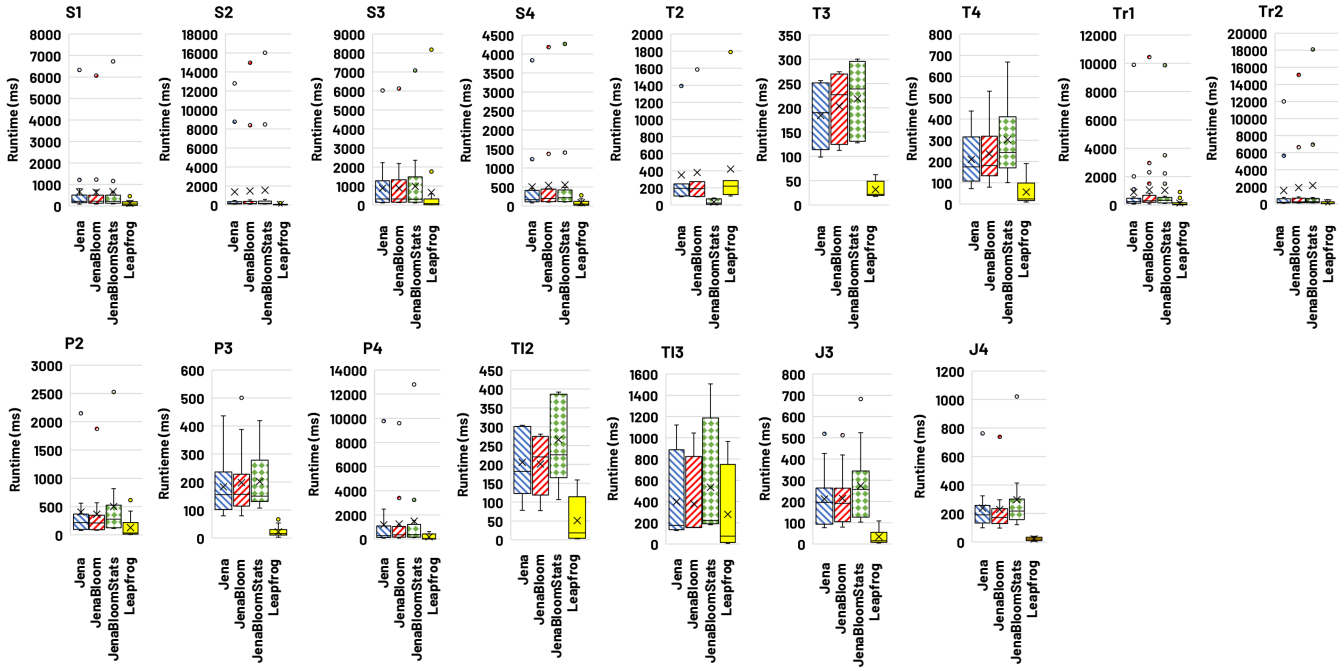

Figure 8: Performance on *BGPs* queries.

```
PREFIX wdp: <http://www.wikidata.org/prop/direct/>
SELECT (COUNT(*) AS ?count) WHERE {
    ?s wdp:P1435 ?o1 .
    ?s wdp:P1435 ?o2 .
    ?o1 wdp:P279 ?o2
}
```

Listing 3: Wikidata SPARQL query.

```
PREFIX wdp: <http://www.wikidata.org/prop/direct/>
PREFIX wdq: <http://www.wikidata.org/entity/>
SELECT (COUNT(*) AS ?count) WHERE {
    wdq:Q7015 ?p ?x2 .
    ?x2 wdp:P155 ?x3 .
    ?x3 wdp:P1204 ?x4 .
    ?x4 wdp:P156 wdq:Q7015
}
```

Listing 4: Wikidata SPARQL query.

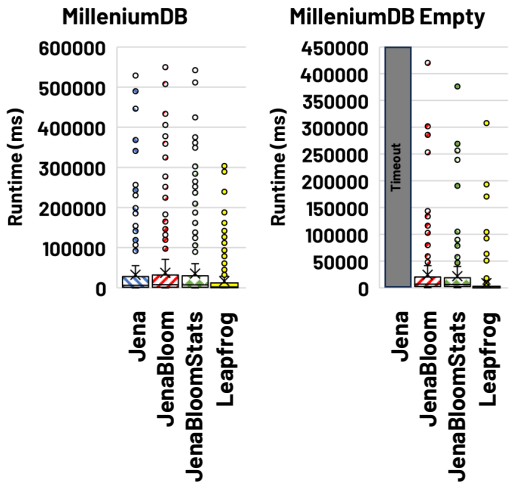

Figure 9: Performance on *MilleniumDB* queries.

We show in Figure 6 the runtimes of all of the approaches for different result set sizes. Unfortunately, the pattern shows that JenaBloom generally is slower than Jena no matter the result set size. JenaBloom using statistics is even slower than JenaBloom due to the approach being too naive, as previously explained.

6 CONCLUSION AND FUTURE WORK

We have explored the application of Bloom filters to perform existence check when joining triple patterns. We have applied predicate statistics to determine which triple patterns should be evaluated with BF existence check. We summarize our findings and future work as follows:

Finding 1: Nested-loop-join is a heavy computation. This comes to display for some of the complex queries. Specifically in those returning empty result sets, where nested-loop-join times out. Faster query processing using BF for existence checking or worst-case optimal join can successfully execute these queries.

Finding 2: Query processing with BF existence check under-performs for the majority of queries, even for those that are specifically designed to maximize existence checks. This is due to the fraction of BF positives being too large for the majority of our benchmark

queries, and hence, BF existence check often must verify its result using disk-based indexes because of the risk of false positives. We express the requirement of fraction of BF negatives in Equation 5.
Finding 3: We observe, as expected, that worst-case optimal join out-performs nested-loop-join. Nested-loop-join with BF existence check can compete with worst-case optimal join for a minority of queries.
Finding 4: The statistical approach to determine the application of BF existence check is too simplistic and does not perform well in determining which triple patterns are suited for BF existence check. This is due to the approach not considering contextual query information in join operations that might impact the probability of BF negatives.
Future work: We would like to experiment with expanding the usage of BF existence checks to triple patterns containing one additional variable that is not included in the join. This would require a much bigger BF, as we need to store each triple three times for each combination of variable positions in the triple. However, this is not a problem, as BFs are compact in size. We would furthermore like to experiment with building BFs of intermediate results sets during query processing.

REFERENCES

[1] Renzo Angles. 2012. A Comparison of Current Graph Database Models. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. 171–177. https://doi.org/10.1109/ICDEW.2012.31

[2] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *ACM Comput. Surv.* 40, 1, Article 1 (Feb. 2008), 39 pages. https://doi.org/10.1145/1322432.1322433

[3] Medha Atre, Jagannathan Srinivasan, and James A Hendler. 2009. BitMat: A main memory RDF triple store. *Tetherless World Constellation, Rensselar Plytehcnic Institute, Troy NY* (2009).

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[6] Orri Erling and Ivan Mikhailov. 2010. *Virtuoso: RDF Support in a Native RDBMS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 501–519. https://doi.org/10.1007/978-3-642-04329-1_21

[7] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4, Article 71 (jul 2021), 37 pages. https://doi.org/10.1145/3447772

[8] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer International Publishing, Cham, 258–275.

[9] Yu Hua, Bin Xiao, Bharadwaj Veeravalli, and Dan Feng. 2012. Locality-Sensitive Bloom Filter for Approximate Membership Query. *IEEE Trans. Comput.* 61, 6 (2012), 817–830. https://doi.org/10.1109/TC.2011.108

[10] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[11] Rishabh Kumar, Hari Prasanna P, Mukund Rungta, Swetha Kashinath Phuleker, Hemant Tiwari, and Vanraj Vala. 2021. FBF: Bloom Filter for Fuzzy Membership Queries on Strings. In *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*. 25–32. https://doi.org/10.1109/ICSC50631.2021.00010

[12] Ora Lassila, Ralph R Swick, et al. 1998. Resource description framework (RDF) model and syntax specification. (1998).

[13] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-Based Graph Database Evaluation. *Proc. VLDB Endow.* 12, 4 (dec 2018), 390–403. https://doi.org/10.14778/3297753.3297759

[14] Yung-Feng Lu and Sheng-Shang Ye. 2012. A Multi-dimension Hash index design for main-memory RFID database applications. In *2012 International Conference on Information Security and Intelligent Control*. 61–64. https://doi.org/10.1109/ISIC.2012.6449708

[15] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2019. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2019), 1912–1949. https://doi.org/10.1109/COMST.2018.2889329

[16] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (2010), 91–113.

[17] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-Scale Knowledge Graphs: Lessons and Challenges. *Commun. ACM* 62, 8 (jul 2019), 36–43. https://doi.org/10.1145/3331166

[18] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1099–1114. https://doi.org/10.1145/3318464.3389702

[19] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL query language for RDF. W3C recommendation, W3C. *URL: http://www. w3. org/TR/rdf-sparql-query* (2008).

[20] Jiangbo Qian, Zhipeng Huang, Qiang Zhu, and Huahui Chen. 2018. Hamming Metric Multi-Granularity Locality-Sensitive Bloom Filter. *IEEE/ACM Transactions on Networking* 26, 4 (2018), 1660–1673. https://doi.org/10.1109/TNET.2018.2850536

[21] Jiangbo Qian, Qiang Zhu, and Huahui Chen. 2015. Multi-Granularity Locality-Sensitive Bloom Filter. *IEEE Trans. Comput.* 64, 12 (2015), 3500–3514. https://doi.org/10.1109/TC.2015.2401011

[22] Tomer Sagi, Matteo Lissandrini, Torben Bach Pedersen, and Katja Hose. 2022. A design space for RDF data representations. *The VLDB Journal* 31, 2 (2022), 347–373.

[23] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. 2020. *Database system concepts*. Vol. 7. McGraw-Hill New York.

[24] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review* 35, 4 (2005), 181–192.

[25] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2021. MillenniumDB: A Persistent, Open-Source, Graph Database. *CoRR* abs/2111.01540 (2021). arXiv:2111.01540 https://arxiv.org/abs/2111.01540

[26] W3C. 2013. *SPARQL Query Language for RDF*. https://www.w3.org/TR/rdf-sparql-query/

[27] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proc. VLDB Endow.* 1, 1 (aug 2008), 1008–1019. https://doi.org/10.14778/1453856.1453965

[28] Peng Zhang, Lichao Xing, Ninggou Yang, Guolin Tan, Qingyun Liu, and Chuang Zhang. 2018. Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. 840–847. https://doi.org/10.1109/BDCloud.2018.00125