Noname manuscript No.
(will be inserted by the editor)

Building Self-Clustering RDF Databases Using Tunable-LSH

Güneş Aluç, M. Tamer Özsu, Khuzaima Daudjee

the date of receipt and acceptance should be inserted later

Abstract The Resource Description Framework (RDF) is a W3C standard for representing graph-structured data, and SPARQL is the standard query language for RDF. Recent advances in information extraction, linked data management and the Semantic Web have led to a rapid increase in both the volume and the variety of RDF data that are publicly available. As businesses start to capitalize on RDF data, RDF data management systems are being exposed to workloads that are far more diverse and dynamic than what they were designed to handle. Consequently, there is a growing need for developing workload-adaptive and self-tuning RDF data management systems. To realize this objective, we introduce a fast and efficient method for dynamically clustering records in an RDF data management system. Specifically, we assume nothing about the workload upfront, but as SPARQL queries are executed, we keep track of records that are co-accessed by the queries in the workload and physically cluster them. To decide dynamically and in constant-time where a record needs to be placed in the storage system, we develop a new locality-sensitive hashing (LSH) scheme, Tunable-LSH. Using Tunable-LSH, records that are co-accessed across similar sets of queries can be hashed to the same or nearby physical pages in the storage system. What sets Tunable-LSH apart from existing LSH schemes is that it can auto-tune to achieve the aforementioned clustering objective with high accuracy even when the workloads change. Experimental evaluation of Tunable-LSH in an RDF data management system as well as in a standalone hashtable shows end-to-end performance gains over existing solutions.

Address(es) of author(s) should be given

1 Introduction

Physical data organization plays an important role in the performance tuning of database management systems. A particularly important problem is clustering records in the storage system that are frequently co-accessed by queries in a workload. Suboptimal clustering has negative performance implications due to random I/O and cache stalls [7]. This problem has received attention in the context of SQL databases and has led to the introduction of tuning advisors that work either in an *offline* [6, 78] or *online* fashion (i.e., self-tuning databases) [31].

In this paper, we address the problem in the context of RDF data management systems. Due to the diversity of applications on the World Wide Web, SPARQL workloads that RDF data management systems service are far more dynamic than conventional SQL workloads [17, 56]. First, on the Web, queries are influenced by real-life events, which can be highly unpredictable [17, 56, 71]. Second, hotspots in RDF, which denote the RDF resources that are frequently queried, can have fluctuating phases of popularity. For example, an analysis over real SPARQL query logs reveal that in one week intervals before, during and after a conference, the popularity of the RDF resources related to that conference can peak and then drop significantly [56]. Third, our attempts at modeling these fluctuations over a collection of real SPARQL workloads [21] using a measure called *volatility* [36] have revealed that the volatility of RDF resources follow normal-like distribution [70]. That is, while some RDF resources are queried frequently and this frequency does not fluctuate much over time (i.e., low volatility), for others this fluctuation can be very high (i.e., high volatility).

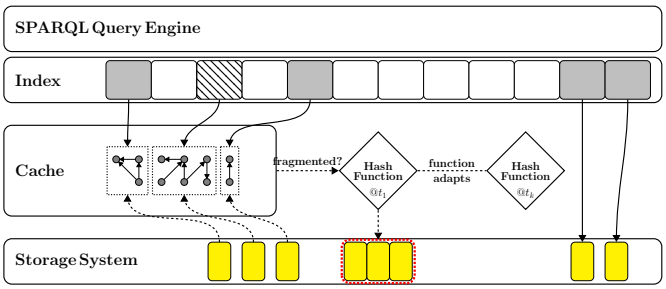


Fig. 1: Adaptive record placement using a combination of adaptive hashing and caching.

For the second class of queries with high volatility, tuning techniques for RDF data management systems are in their infancy, and relational solutions are not directly applicable. More specifically, depending on the workload, it might be necessary to completely change the underlying physical representation in an RDF data management system, such as by dynamically switching from a row-oriented representation to a columnar representation [13]. On the other hand, existing solutions are either *offline* requiring the workload to be known upfront [39, 48], or *online* for which the tuning techniques work well only when the schema changes are minor [27]. Consequently, with the increasing demand to support highly dynamic workloads in RDF [17, 56], there is a growing need to develop more adaptive tuning solutions in which records in an RDF database can be dynamically and continuously clustered based on the current workload.

Whenever a SPARQL query is executed, there is an opportunity to observe how records in an RDF database are being utilized. This information about query access patterns can be used to dynamically cluster records in the storage system. The ability to do this dynamically is important in RDF systems because of the high variability and dynamism present in SPARQL workloads [17, 56]. While this problem has been studied as physical clustering [60] and distribution design [29], the highly dynamic nature of the queries over RDF data introduces new challenges. First, traditional algorithms are offline, and since clustering is an NP-hard problem where most approximations have quadratic complexity [55], they are not suitable for online database clustering. Instead, techniques are needed with similar clustering objectives but that have linear running time. Second, systems are typically expected to execute most queries in subseconds [65], leaving only fractions of a second to update their physical data structures (e.g., dynamically moving records across the storage system).

We address the aforementioned issues by making two contributions. First, as shown in Fig. 1, instead of clustering the whole database, we cluster only the "warm" portions of the database by relying on the admission policy of the existing database cache. Second, we develop a novel self-tuning locality-sensitive hash (LSH) function, namely, Tunable-LSH to decide in constant-time where in the storage system to place a record. Tunable-LSH has two important properties: First, it strives to ensure that (i) records with *similar* utilization patterns (i.e., those records that are co-accessed across similar sets of queries) are mapped as much as possible to the same pages in the storage system, while (ii) minimizing the number of records with *dissimilar* utilization patterns that are falsely mapped to the same page. Second, unlike conventional LSH [38, 53], Tunable-LSH can auto-tune so as to achieve the aforementioned clustering objectives with high accuracy even when the workload changes.

These ideas are illustrated in Fig. 1. Let us assume that initially, the records in a database are *not* clustered according to any particular workload. Therefore, the performance of the system is suboptimal. However, every time records are fetched from the storage system, there is an opportunity to bring together into a single page those records that are co-accessed but are fragmented across the storage system. Tunable-LSH achieves this goal with minimal overhead. Furthermore, Tunable-LSH is continuously updated to reflect changes in the workload characteristics. Consequently, as more queries are executed, records in the database become more clustered, thereby, improving performance.

The paper is organized as follows: Section 2 discusses related work. Section 3 gives a conceptual description of the problem. Section 4 describes the overview of our approach while Section 5 provides the details. In Section 6, we experimentally evaluate our techniques. Finally, we discuss conclusions and future work in Section 7.

2 Related Work

Locality-sensitive hashing (LSH) [38, 53] has been used in various contexts such as nearest neighbour search [16, 18, 34, 49, 53, 73], Web document clustering [25, 26] and query plan caching [11]. In this paper, we use LSH in the physical design of RDF databases. While multiple families of LSH functions have been developed [26, 30, 32, 38, 53], these functions assume that the input distribution is either uniform or static. In contrast, Tunable-LSH can continuously adapt to changes in the input distribution to achieve higher accuracy, which translates to

adapting to changes in the workload query access patterns in the context of RDF databases.

Physical design has been a topic of ongoing discussion in the world of RDF and SPARQL [4, 13, 72, 75]. One option is to represent data in a single large table [28] and build clustered indexes, where each index implements a different sort order [46,67,74]. TripleBit [76] takes this idea further and implements indexes using bit-vectors. QLever [19] adds text indexes on top. It has also been argued that grouping data can help improve performance [4, 72]. For this reason, multiple physical representations have been developed: in the *group-by-predicate* representation, the database is vertically partitioned and the tables are stored in a column-store [4]; in the *group-by-entity* representation, implicit relationships within the database are discovered (either manually [75] or automatically [24, 50]), and the RDF data are mapped to a relational database or into native storage [47]; and in the *group-by-vertex* representation, the inherent graph-structure of RDF is preserved, whereby data can be grouped by the vertices in the graph [79]. TriAD [41] is a distributed RDF engine whereby graph-locality is preserved within each distributed node. These workload-oblivious representations have issues for different types of queries due to reasons such as fragmented data, unnecessarily large intermediate result tuples generated during query evaluation and/or suboptimal index pruning [13].

To address some of these issues, workload-aware techniques have been proposed [39, 48]. For example, view materialization techniques have been implemented for RDF over relational engines [39, 61]. However, these materialized views are difficult to adapt to changing workloads for reasons discussed in Section 1. Workload-aware distribution techniques have also been developed for RDF [48] and implemented in systems such as Partout [37] and WARP [48], but these systems are not runtime-adaptive. With Tunable-LSH, we aim to address the problem adaptively, by clustering fragmented records in the database based on the workload.

Although there are self-tuning SQL databases [31, 51, 52] and techniques for automatic schema design in SQL [6, 20, 60, 78], these techniques are not directly applicable to RDF. In RDF, the advised changes to the underlying physical schema can be drastic, for example, requiring the system to switch from a row-oriented representation to a columnar one, all at runtime, which are hard to achieve using existing techniques. Consequently, there have been efforts in designing workload-adaptive and self-tuning RDF data management systems [8,9,13,14,44,68,69]. In H2RDF [68], the choice between centralized versus distributed execution is made adaptively. A mechanism for adaptively caching partial results is introduced in [69]. AdPart [8, 44] is a distributed, workload-adaptive RDF data management system that is closely related. AdPart distributes triples by hashing on triples' subjects. The hash function is static and not locality-sensitive (unlike Tunable-LSH); therefore, AdPart relies on dynamic re-distribution and replication of frequently accessed data to adapt to the workload (the underlying physical layout is fixed within each node). To this end, AdPart and our work offer complementary solutions: AdPart would benefit from Tunable-LSH in determining which triples should be clustered together, and our work would benefit from adaptive replication.

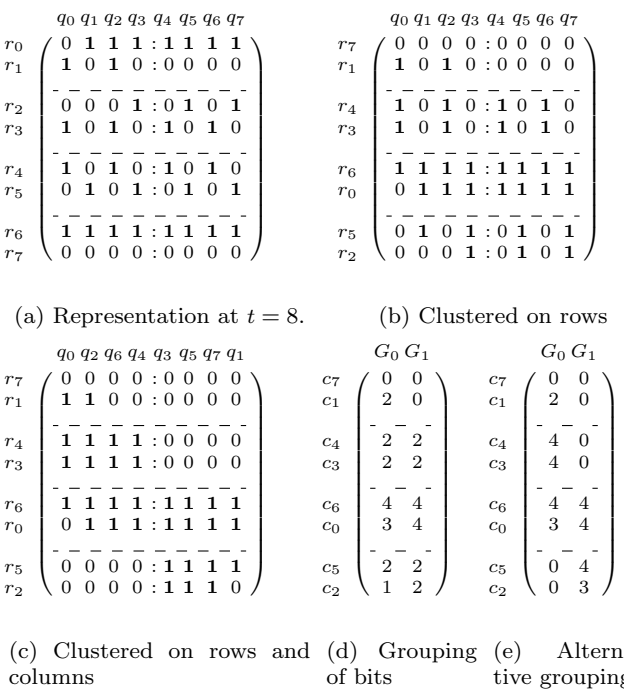


Fig. 2: Matrix representation of query access patterns.

3 Preliminaries

Given a sequence of database records that represent the records' serialization order in the storage system, the access patterns of a query can conceptually be represented as a bit vector, where a bit is set to 1 if the corresponding record in the sequence is accessed by that query. We call this bit vector a *query access vector* ($\mathbf{q}$).

Depending on the system, a record may denote a single RDF triple (i.e., the atomic unit of information in RDF) or a collection of RDF triples. Our conceptual model is applicable either way.

As more queries are executed, their query access vectors can be accumulated column-by-column in a ma-

trix, as shown in Fig. 2a. We call this matrix a *query access matrix*. For presentation, let us assume that queries are numbered according to their order of execution by the RDF data management system.

Each row of the query access matrix constitutes what we call a *record utilization vector* ($\mathbf{r}$), which represents the set of queries that access record r. As a convention, to distinguish between a query and its access vector (likewise, a record and its utilization vector), we use the symbols q and $\mathbf{q}$ (likewise, r and $\mathbf{r}$), respectively. The complete list of symbols are given in Table 1.

To model the memory hierarchy, we use an additional notation in the matrix representation: records that are physically stored together on the same disk or memory page should be grouped together in the query access matrix. For example, Fig. 2a and Fig. 2b represent two alternative ways in which the records in an RDF database can be clustered (groups are separated by horizontal dashed lines). Even though both figures depict essentially the same query access patterns, the physical organization in Fig. 2b is superior, because in Fig. 2a, most queries require access to 4 pages each, whereas in Fig. 2b, the number of accesses is reduced by almost half.

Given a sequence of queries and the number of pages in the storage system, our objective is to *store records having similar utilization vectors together so as to minimize the total number of page accesses*. To determine the similarity between record utilization vectors, we exploit the following property: two records are co-accessed by a query if both of the corresponding bits in that query's access vector are set to 1. Extending this concept to a set of queries, we say that two records are co-accessed across multiple queries if the corresponding bits in the record utilization vectors are set to 1 for all the queries in the set. For example, according to Fig. 2a, records r_1 and r_3 are co-accessed by queries q_0 and q_2, and records r_0 and r_6 are co-accessed across the queries q_1–q_7.

Given a sequence of queries, it is possible that a pair of records are not co-accessed in *all* of the queries. Therefore, to measure the extent to which a pair of records are co-accessed, we rely on their Hamming distance [43]. Specifically, given two record utilization vectors for the same sequence of queries, their Hamming distance—denoted as $\delta(\mathbf{q}_x, \mathbf{q}_y)$—is defined as the minimum number of substitutions necessary to make the two bit vectors the same [43].[1] Hence, the smaller the Hamming distance between a pair of records, the greater the extent to which they are co-accessed.

[1] The Hamming distance between two record utilization vectors is equal to their edit distance [59], as well as the Manhattan distance [57] between these two vectors in l_1 norm.

Consider the record utilization vectors $\mathbf{r_0}$, $\mathbf{r_2}$, $\mathbf{r_5}$ and $\mathbf{r_6}$ across the query sequence q_0–q_7 in Fig. 2a. The pairwise Hamming distances are as follows: $\delta(r_0, r_6) = 1$, $\delta(r_2, r_5) = 1$, $\delta(r_0, r_5) = 3$, $\delta(r_0, r_2) = 4$, $\delta(r_5, r_6) = 4$ and $\delta(r_2, r_6) = 5$. Consequently, to achieve better physical clustering, r_0 and r_6 should be stored together, as are r_2 and r_5, while r_0 and r_6 should be kept apart from r_2 and r_5.

4 Overview of Tunable-LSH

The dynamic nature of queries over RDF data necessitate a solution different from existing clustering algorithms [13]. That is, while conventional clustering algorithms [55] might be perfectly applicable for the *offline* tuning of a database, in an *online* scenario, what is needed is an algorithm that clusters records on-the-fly and within microseconds. Clustering is an NP-hard problem [55], and most approximations take at least quadratic time [5]. In contrast, Tunable-LSH is a self-tuning locality-sensitive hash (LSH) function that can approximate clustering in linear time. Tunable-LSH is used as follows.

As records are fetched from the storage system, we keep track of records that are accessed by the same query but are fragmented across the pages in the storage system. Then, we use Tunable-LSH to decide, in constant-time, how a fragmented record needs to be clustered in the storage system (cf., Fig. 1). Furthermore, we develop methods to continuously autotune this LSH function to adapt to changing query access patterns that are encountered while executing a workload. This way, Tunable-LSH can achieve much higher clustering accuracy than conventional LSH techniques, which are static.

Let $\mathbb{Z}_{\alpha \dots \beta}$ denote the set of integers in the interval $[\alpha, \beta]$, and let $\mathbb{Z}^n_{\alpha \dots \beta}$ denote the n-fold Cartesian product:

$$\underbrace{\mathbb{Z}_{\alpha \dots \beta} \times \cdots \times \mathbb{Z}_{\alpha \dots \beta}}_{n}.$$

Furthermore, let us assume that we are given a non-injective, surjective function $f : \mathbb{Z}_{0 \dots (k-1)} \to \mathbb{Z}_{0 \dots (b-1)}$, where $b \ll k$, and for all $y \in \mathbb{Z}_{0 \dots (b-1)}$, it holds that

$$\Big|\{x : f(x) = y\}\Big| \leq \left\lceil \frac{k}{b} \right\rceil.$$

In other words, f is a hash function with the property that given k input values and b possible outcomes, no more than $\lceil \frac{k}{b} \rceil$ values in the domain of the function will be hashed to the same value (Section 5.2 discusses

	Symbol	Description
Constants	ω	database size (i.e., number of records)
	ϵ	number of pages in the storage system
	k	maximum no. of query access vectors that can be stored
	b	number of entries in each record utilization counter
	t	current time
Data structures	$\mathbf{q}$	query access vector (contains ω bits)
	$\mathbf{r}$	record utilization vector (contains k bits)
	$\mathbf{c}$	record utilization counter (contains b entries)
	$\mathbf{P}$	depending on the context, a point in a k-dimensional or b-dimensional (Taxicab) space
	$M_{\omega\times k}$	query access matrix; contains the last k most representative query access vectors (in columns), or equivalently, ω record utilization vectors (in rows)
	$C_{\omega\times b}$	frequency matrix; represents record utilization frequency over b groups of query access vectors
Accessors	$q[i]$	value of the i^{th} bit in query access vector $\mathbf{q}$
	$r[i]$	value of the i^{th} bit in record utilization vector $\mathbf{r}$
	$c[i]$	value of the i^{th} entry in record utilization counter $\mathbf{c}$
	$P[i]$	value of the i^{th} coordinate in point $\mathbf{P}$
	$M[i][j]$	value of the i^{th} row and j^{th} column in matrix
	$C[i][j]$	value of the i^{th} row and j^{th} column in matrix
Distances	$\delta(\mathbf{r_x}, \mathbf{r_y})$	Hamming distance between two record utilization vectors
	$\delta^H(\mathbf{q_x}, \mathbf{q_y})$	MIN-HASH distance between two query access vectors
	$\delta^M(\mathbf{P_x}, \mathbf{P_y})$	Manhattan distance between two points

Table 1: Symbols used throughout the manuscript

in more detail how f can be constructed).[2] Then, we define TUNABLE-LSH as $h : \mathbb{Z}^k_{0\cdots 1} \to \mathbb{Z}_{0\cdots(\epsilon-1)}$, where ϵ represents the number of pages in the storage system. More specifically, h is defined as a composition of two functions h_1 and h_2.

Definition 1 (Tunable-LSH)

Let

$$\mathbf{r} = (r[0], \ldots, r[k-1]) \in \mathbb{Z}^k_{0\cdots 1}, \text{ and}$$
$$\mathbf{c} = (c[0], \ldots, c[b-1]) \in \mathbb{Z}^b_{0\cdots\lceil\frac{k}{b}\rceil}.$$

Then, an adaptive LSH function h is defined as

$$h = \mathbf{h_2} \circ \mathbf{h_1}$$

where

$\mathbf{h_1} : \mathbb{Z}^k_{0\cdots 1} \to \mathbb{Z}^b_{0\cdots\lceil\frac{k}{b}\rceil}$, where $h_1(\mathbf{r}) = \mathbf{c}$ iff

$$\forall y\ c[y] = \sum_{x=0}^{k-1} \begin{cases} r[x] : f(x) = y \\ 0 \quad\ : f(x) \neq y \end{cases}$$

$\mathbf{h_2} : \mathbb{Z}^b_{0\cdots\lceil\frac{k}{b}\rceil} \to \mathbb{Z}_{0\cdots(\epsilon-1)}$, where $h_2(\mathbf{c}) = v$ and v is the coordinate of $\mathbf{c}$ (rounded to the nearest integer) on a space-filling curve [64] of length ϵ that covers $\mathbb{Z}^b_{0\cdots\lceil\frac{k}{b}\rceil}$.

According to Def. 1, h is constructed as follows:

1. Using a hash function f (which can be treated as a black box for the moment), a record utilization vector $\mathbf{r}$ with k bits is divided into b disjoint segments $\mathbf{r}_0, \ldots, \mathbf{r}_{b-1}$ such that $\mathbf{r}_0, \ldots, \mathbf{r}_{b-1}$ contain all the bits in $\mathbf{r}$, and each $\mathbf{r}_i \in \{\mathbf{r}_0, \ldots, \mathbf{r}_{b-1}\}$ has at most $\lceil\frac{k}{b}\rceil$ bits. Then, a record utilization counter $\mathbf{c}$ with b entries is computed such that the i^{th} entry of $\mathbf{c}$ (i.e., $c[i]$) contains the number of 1-bits in $\mathbf{r}_i$. Without loss of generality, a record utilization counter $\mathbf{c}$ can be represented as a b-dimensional point in the coordinate system $\mathbb{Z}^b_{0\cdots\lceil\frac{k}{b}\rceil}$.

[2] This uniformity condition simplifies the sensitivity analysis of TUNABLE-LSH, but it is not a requirement from an algorithmic point of view. Relaxing this condition is left as future work.

Symbol	*Description*
begin	natural number between $0 \dots (k-1)$, initial value is 0
size	natural number between $0 \dots (k-1)$, keeps track of the number of query access vectors that are currently being maintained, initial value is 0
$H_{k\times?}$	matrix that contains MIN-HASH values for each query access vector
$S[]$	array of *vector*(s), one for each MDS query point, that pairs each MDS query point with a *random* subset of points
$N[]$	array of *max-heap*(s), one for each MDS query point, that pairs each MDS query point with a set of *neighboring* points
$X[]$	array of *float*(s), represents the coordinate (single dimensional) of each MDS query point
$V[]$	array of *float*(s), represents the current (directional) velocity of each MDS query point

Table 2: Data structures referenced in algorithms

	Coordinates			
	Decimal	Binary	z-value	Hash Value
$\mathbf{c}_7$	$(0,0)$	$(000,000)$	000000	00xxxx
$\mathbf{c}_1$	$(2,0)$	$(010,000)$	001000	00xxxx
$\mathbf{c}_4$	$(2,2)$	$(010,010)$	001100	00xxxx
$\mathbf{c}_3$	$(2,2)$	$(010,010)$	001100	00xxxx
$\mathbf{c}_6$	$(4,4)$	$(100,100)$	110000	11xxxx
$\mathbf{c}_0$	$(3,4)$	$(011,100)$	011010	01xxxx
$\mathbf{c}_5$	$(2,2)$	$(010,010)$	001100	00xxxx
$\mathbf{c}_2$	$(1,2)$	$(001,010)$	000110	00xxxx

Table 3: Illustration of h_2 in Definition 1

2. The final hash value is computed by computing the z-value [35, 64] of the points in $\mathbb{Z}^b_{0\dots\lceil\frac{k}{b}\rceil}$, and dropping off the last m bits from the produced z-values, where $m = b(\lceil\log_2\lceil\frac{k}{b}\rceil\rceil + 1) - \lceil\log_2\epsilon\rceil$.

Consider the record utilization vectors $\mathbf{r}_0 \dots \mathbf{r}_7$ in Fig. 2b. Let us assume that a hash function f divides the bits in the record utilization vectors into two groups such that columns $q_0 \dots q_3$ appear in one group and $q_4 \dots q_7$ appear in another. Then, the record utilization counters depicted in Fig. 2d can be generated according to h_1 in Definition 1. h_2 takes these record utilization counters and computes their z-values. To illustrate that process (cf., Table 3), we represent the record utilization counters as points in a two-dimensional space (where b – the number of groups h_1 has generated – determines the dimensionality) and rely on the coordinates of the points in binary notation. Computing the z-values is as simple as interleaving the bits from each dimension; that is, we pick out a red bit and then a blue, and repeat this process from left to right until all bits are consumed. To compute the final hash values, we will mask out the least significant bits from the z-values. Exactly how many bits we mask out depends on the total number of distinct values the hash function is allowed to generate, which is determined by the number of pages in the system. In this example, we assume that we will be distributing the records across four pages in total, hence, we mask out the last four bits from each z-value to end up with two-bit hash values.

In Section 5.1, we show that TUNABLE-LSH that maps k-dimensional record utilization vectors to natural numbers in the interval $[0, \dots, \epsilon - 1]$ is locality-sensitive, with two important implications: (i) records with similar record utilization vectors (i.e., small Hamming distances) are likely to be hashed to the same value, while (ii) records with dissimilar record utilization vectors are likely to be separated. Therefore, the problem of clustering records in the storage system can be approximated using TUNABLE-LSH such that clustering n records takes $O(n)$ time.

The quality of TUNABLE-LSH, that is, how well it approximates the original Hamming distances, depends on two factors: (i) the characteristics of the workload so far, which is reflected by the bit distribution in the record utilization vectors, and (ii) the choice of f. In Section 5.2, we demonstrate that f can be tuned to adapt to the changing patterns in record utilization vectors to maintain the approximation quality of TUNABLE-LSH at a steady and high level.

Algorithms 1–3 present our approach for computing the outcome of TUNABLE-LSH and for incrementally tuning the LSH function after every query execution. Note that we have two design considerations: (i) tuning

Algorithm 1 Initialize

Ensure:
Record utilization counters are allocated and initialized

```
1: procedure INITIALIZE()
2:     construct int C[ω][2b]      ▷ For simplicity, C is allocated statically; however, in practice, it can be allocated dynamically to reduce memory footprint.
3:     for all i ∈ (0, ..., ω − 1) do
4:         for all j ∈ (0, ..., 2b − 1) do
5:             C[i][j] ← 0
6:         end for
7:     end for
8: end procedure
```

Algorithm 2 Tune

Require:
$\mathbf{q_t}$: query access vector produced at time t
Ensure:
Underlying data structures are updated and f is tuned such that the LSH function maintains a steady approximation quality

```
1: procedure TUNE(q_t)
2:     RECONFIGURE-F(q_t)
3:     for all i ∈ POSITIONAL(q_t) do
4:         loc ← f(t)
5:         if loc < (shift % b) then
6:             loc += b
7:         end if
8:         C[i][loc]++      ▷ Increment record utilization counters based on the new query access pattern
9:         if t%⌈k/b⌉ = 0 then      ▷ Reset "old" counters
10:            shift++
11:            C[i][(shift+b)%2b] ← 0
12:        end if
13:    end for
14: end procedure
```

Algorithm 3 Hash

Require:
id: id of record whose hash is being computed
Ensure:
Hash value is returned

```
1: procedure HASH(id)
2:     return Z-VALUE(C[id])      ▷ Apply h_2
3: end procedure
```

should take constant-time, otherwise, there is no point in using a function, (ii) the memory footprint should be low because it would be desirable to maximize the allocation of memory to core database functionality. Consequently, instead of relying on record utilization vectors, the algorithm computes and incrementally maintains record utilization counters (cf., Algorithm 1), which are much easier to maintain and have a much smaller memory footprint due to $b \ll k$. Then, whenever there is a need to compute the outcome of the LSH function for a given record, the HASH procedure is called with the *id* of the record, which in turn relies on h_2 to compute the hash value (cf., Algorithm 3).

The TUNE procedure looks at the next query access vector, and updates f (line 2), which will be discussed in more detail in Section 5.2. Then it computes positions of records that have been accessed by that query (line 3), and increments the corresponding entries in the utilization counters of those records that have been accessed (line 8). To determine which entry to increment, the algorithm relies on h_1, hence, $f(t)$ (cf., Def. 1) and a shifting scheme. In line 11, old entries in record utilization counters are reset based on an approach that we discuss in Section 5.3. In that section we also discuss the shifting scheme.

5 Details of Tunable-LSH and Optimizations

This section is structured as follows: Section 5.1 shows that TUNABLE-LSH has the properties of a locality-sensitive hashing scheme. Section 5.2 describes our approach for tuning f based on the most recent query access patterns, and Section 5.3 explains how old bits are removed from record utilization counters. Section 5.4 discusses potential future work to support insertions in TUNABLE-LSH.

5.1 Properties of Tunable-LSH

This section discusses the locality-sensitive properties of $h = h_2 \circ h_1$ and demonstrates that h can be used for clustering the records. First, the relationship between record utilization vectors and the record utilization counters that are obtained by applying h_1 is shown.

Theorem 1 (Distance Bounds)
Given a pair of record utilization vectors $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *with size* k*, let* $\mathbf{c}_1$ *and* $\mathbf{c}_2$ *denote two record utilization counters with size* b *such that* $\mathbf{c}_1 = h_1(\mathbf{r}_1)$ *and*

$\mathbf{c}_2 = h_1(\mathbf{r}_2)$ *(cf., Definition 1). Furthermore, let* $c_1[i]$ *and* $c_2[i]$ *denote the* i^{th} *entry in* $\mathbf{c}_1$ *and* $\mathbf{c}_2$*, respectively. Then,*

$$\delta(\mathbf{r_1}, \mathbf{r_2}) \geq \sum_{i=0}^{b-1} \left| c_1[i] - c_2[i] \right| \tag{1}$$

where $\delta(\mathbf{r_1}, \mathbf{r_2})$ *represents the Hamming distance between* $\mathbf{r}_1$ *and* $\mathbf{r}_2$*.*

Proof. *It is possible to prove Theorem 1 by induction on* b*.*

Base case*: Theorem 1 holds when* $b = 1$*. According to Definition 1, when* $b = 1$*,* $c_1[0]$ *and* $c_2[0]$ *correspond to the total number of* 1*-bits in* $\mathbf{r}_1$ *and* $\mathbf{r}_2$*, respectively. Note that the Hamming distance between* $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *will be smallest if and only if these two record utilization vectors are aligned on as many* 1*-bits as possible. In that case, they will differ in only* $\left|c_1[0] - c_2[0]\right|$ *bits, which corresponds to their Hamming distance. Consequently, Equation 1 holds for* $b = 1$*.*

Inductive step*: It needs to be shown that if Equation 1 holds for* $b \leq \alpha$*, where* α *is a natural number greater than or equal to* 1*, then it must also hold for* $b = \alpha + 1$*.*

Let $\Pi_f(\mathbf{r}, g)$ *denote a record utilization vector* $r' = (r'[0], \ldots, r'[k-1])$ *such that for all* $i \in \{0, \ldots, k-1\}$*,* $r'[i] = r[i]$ *holds if* $f(i) = g$*, and* $r'[i] = 0$ *otherwise. Then,*

$$\delta(\mathbf{r}_1, \mathbf{r}_2) = \sum_{g=0}^{b-1} \delta(\Pi_f(\mathbf{r}_1, g), \Pi_f(\mathbf{r}_2, g)). \tag{2}$$

That is, the Hamming distance between any two record utilization vectors is the summation of their individual Hamming distances within each group of bits that share the same hash value with respect to f*. This property holds because* f *is a (total) function, and* Π_f *masks all the irrelevant bits. As an abbreviation, let*

$$\delta_g = \delta(\Pi_f(\mathbf{r}_1, g), \Pi_f(\mathbf{r}_2, g)).$$

Then, due to the same reasoning as in the base case, for $g = \alpha$*, the following equation holds:*

$$\delta_\alpha(\mathbf{r_1}, \mathbf{r_2}) \geq \left| c_1[\alpha] - c_2[\alpha] \right| \tag{3}$$

Consequently, due to the additive property in Equation 2, Equation 1 holds also for $b = \alpha + 1$*. Thus, by induction, Theorem 1 holds.* □

Theorem 1 suggests that the Hamming distance between any two record utilization vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ can be approximated using record utilization counters $\mathbf{c_1} = h_1(\mathbf{r_1})$ and $\mathbf{c_2} = h_1(\mathbf{r_2})$ because Equation 1 provides a lower bound on $\delta(\mathbf{r_1}, \mathbf{r_2})$. In fact, the right-hand side of Equation 1 is equal to the Manhattan distance [57] between $\mathbf{c_1}$ and $\mathbf{c_2}$ in $\mathbb{Z}^b_{0 \ldots \lceil \frac{k}{b} \rceil}$, and since $\delta(\mathbf{r_1}, \mathbf{r_2})$ is equal to the Manhattan distance between $\mathbf{r_1}$ and $\mathbf{r_2}$ in $\mathbb{Z}^k_{0 \ldots 1}$, it is easy to see that h_1 is a transformation that approximates Manhattan distances. The following corollary captures this property.

Corollary 1 (Distance Approximation) *Given two record utilization vectors* $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *each with size* k*, let* $\mathbf{c_1}$ *and* $\mathbf{c_2}$ *denote two points in the coordinate system* $\mathbb{Z}^b_{0 \ldots \lceil \frac{k}{b} \rceil}$ *such that* $\mathbf{c_1} = h_1(\mathbf{r}_1)$ *and* $\mathbf{c_2} = h_1(\mathbf{r}_2)$ *(cf., Definition 1). Let* $\delta^M(\mathbf{r_1}, \mathbf{r_2})$ *denote the Manhattan distance between* $\mathbf{r_1}$ *and* $\mathbf{r_2}$*, and let* $\delta^M(\mathbf{c_1}, \mathbf{c_2})$ *denote the Manhattan distance between* $\mathbf{c_1}$ *and* $\mathbf{c_2}$*. Then, the following holds:*

$$\delta(\mathbf{r}_1, \mathbf{r}_2) = \delta^M(\mathbf{r_1}, \mathbf{r_2}) \geq \delta^M(\mathbf{c_1}, \mathbf{c_2}) \tag{4}$$

Proof. *Hamming distance in* $\mathbb{Z}^k_{0 \ldots 1}$ *is a special case of Manhattan distance. Furthermore, by definition [57], the right hand side of Equation 1 equals the Manhattan distance* $\delta^M(\mathbf{c_1}, \mathbf{c_2})$*; therefore, Equation 4 holds.* □

Next, we demonstrate that $h = h_2 \circ h_1$ is a locality-sensitive transformation [38,53]. In particular, the definition of locality-sensitiveness by Tao et al. [73] is used, and it is shown that the probability that two record utilization vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ are hashed to the same page increases as the (Manhattan) distance between r_1 and r_2 decreases.

Theorem 2 (Collision Probabilities) *Given a pair of record utilization vectors* $\mathbf{r_1}$ *and* $\mathbf{r_2}$ *with size* k*, let* $\delta^M(\mathbf{r_1}, \mathbf{r_2})$ *denote the Manhattan distance between* $\mathbf{r_1}$ *and* $\mathbf{r_2}$*. Furthermore, let* m *denote the number of rightmost bits that are dropped by* h_2*. When* $b = 1$ *(i.e., the size of the record utilization counters produced by* h_1*), the probability that* the pair of record utilization vectors will be hashed to the same value by $h_2 \circ h_1$ provided that their initial Manhattan distance is x *is given by the following formula:*

$$\Pr\Big(h_2 \circ h_1(\mathbf{r_1}) = h_2 \circ h_1(\mathbf{r_2}) \;\Big|\; \delta^M(\mathbf{r_1}, \mathbf{r_2}) = x\Big) = \frac{\sum_{a=0}^{x} \sum_{\Delta=0}^{k-x} \binom{x}{a} \binom{k}{k-x} \binom{k-x}{\Delta} \rho(\Delta + a, x - 2a, m)}{2^{2k}} \tag{5}$$

where $\rho(x, y, m) : (\mathbb{Z}_{0 \ldots \infty}, \mathbb{Z}_{-\infty \ldots \infty}, \mathbb{Z}_{0 \ldots \infty}) \rightarrow \{0, 1\}$ *is a function such that*

$$\rho(x, y, m) = \begin{cases} 1 \text{ if } 0 \leq (x \bmod 2^m) + y < 2^m \\ 0 \text{ else} \end{cases}.$$

Proof. *If the Hamming/Manhattan distance between* $\mathbf{r_1}$ *and* $\mathbf{r_2}$ *is* x*, then it means that these two vectors will differ in exactly* x *bits, as shown below.*

$$\mathbf{r_1}: \square\square\square \overbrace{\mathbf{111\ldots1}}^{a} \mathbf{0\cdots000}\square\square\square$$
$$\mathbf{r_2}: \square\square\square \mathbf{000\cdots0} \underbrace{\mathbf{1\ldots111}}_{x-a} \square\square\square$$

Furthermore, if $\mathbf{r_1}$ *has* $\Delta+a$ *bits set to* 1*, then* $\mathbf{r_2}$ *must have* $\Delta+(x-a)$ *bits set to* 1*, where* Δ *denotes the number of matching* 1*-bits between* $\mathbf{r_1}$ *and* $\mathbf{r_2}$*. Note that when* $b=1$*,* $\mathbf{c_1}=h_1(\mathbf{r_1})=(\Delta+a)$ *and* $\mathbf{c_2}=h_1(\mathbf{r_2})=(\Delta+x-a)$*.*

It is easy to see that $a\in\mathbb{Z}_{0\,\ldots\,x}$ *and* $\Delta\in\mathbb{Z}_{0\,\ldots\,k-x}$*. For each value of* a*, the non-matching bits in* $\mathbf{r_1}$ *and* $\mathbf{r_2}$ *can be combined in* $\binom{x}{a}$ *possible ways, and these non-matching bits can be positioned across the* k *bits in* $\binom{k}{k-x}$ *possible ways. Likewise, for each value of* Δ*, those matching* 1*-bits that are counted by* Δ *can be combined in* $\binom{k-x}{\Delta}$ *possible ways, hence, the first three components of the multiplication in the numerator of Equation 5.*

Among the aforementioned combinations, $h_2(\mathbf{c}_1)=h_2(\mathbf{c}_2)$ *will be true if and only if the binary representations of* $\mathbf{c}_1$ *and* $\mathbf{c}_2$ *share the same sequence of bits except for their last* m *bits. This condition will be satisfied if and only if* $\Delta+a$ *and* $\Delta+x-a$ *have the same quotient when divided by* 2^m*. In other words,* $(\Delta+a)\bmod 2^m+(\Delta+x-a)-(\Delta+a)$ *must be greater than or equal to* 0 *or less than* 2^m*, hence, the need to multiply by* ρ *in the numerator of Equation 5.*

Since $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *consist of* k *bits, there can be* $2^k\times 2^k=2^{2k}$ *possible combinations in total, which corresponds to the denominator of Equation 5.* □

Next, Theorem 2 is extended to cases in which $b\geq 2$.

Lemma 1 *Let* $\delta^M(\mathbf{r}_i,\,\mathbf{r}_j)$ *denote the Manhattan distance between any two record utilization vectors* $\mathbf{r}_i$ *and* $\mathbf{r}_j$*, and let* m *denote the number of rightmost bits that are dropped by* h_2*, which is utilized in* h *(cf., Definition 1). Furthermore, let* $\text{PR}_{==}\big(\mathbf{r}_i,\,\mathbf{r}_j,\,x\big)$ *denote the posterior probability that, for any two record utilization vectors* $\mathbf{r}_i$ *and* $\mathbf{r}_j$*,* $h(\mathbf{r}_i)=h(\mathbf{r}_j)$ *provided that* $\delta^M(\mathbf{r}_i,\,\mathbf{r}_j)=x$*. Given any four record utilization vectors* $\mathbf{r}_1$*,* $\mathbf{r}_2$*,* $\mathbf{r}_3$ *and* $\mathbf{r}_4$ *with size* λb *such that*

$$\delta^M(\mathbf{r}_1,\,\mathbf{r}_2)=x \text{ and}$$
$$\delta^M(\mathbf{r}_3,\,\mathbf{r}_4)=\lambda b-x,$$

the following property holds for any $x\leq\frac{\lambda}{2}$ *and* $m=\Upsilon b$*:*

$$\text{PR}_{==}\big(\mathbf{r}_1,\,\mathbf{r}_2,\,x\big)\geq\text{PR}_{==}\big(\mathbf{r}_3,\,\mathbf{r}_4,\,\lambda b-x\big) \tag{6}$$

where $h=h_2\circ h_1$ *(cf., Definition 1),* $b\in\mathbb{Z}_{1\,\ldots\,\infty}$ *denotes the number of entries in the record utilization counters produced by* h_1*,* λ *is an even and positive number, and* $\Upsilon\in\mathbb{Z}_{1\,\ldots\,\lambda-1}$*.*

The proof is lengthy, therefore, only a proof sketch is provided in this section. The complete proof is available in [10].

Proof. *Lemma 1 is proven by induction on* b*.*
Base case*: Equation 6 holds when* $b=1$*. The proof sketch is below:*

B1. It is shown that for any two natural numbers a *and* Δ *such that* $a\leq\frac{x}{2}$*,* $\Delta\leq(k-x)$*,* $\Delta-a>\frac{k-2x}{2}$*, and* $\Delta+a<\frac{k}{2}$*, if* $\rho(a+\Delta,\,x-2a)<\rho(a+\Delta,\,k-x-2\Delta)$*, there exists two natural numbers* a' *and* Δ' *such that*

$$0\leq a'=\Delta-\frac{k-2x}{2}\leq x \text{ and}$$
$$0\leq\Delta'=a+\frac{k-2x}{2}\leq k-x,$$

and the following property holds:

$$\binom{x}{a}\binom{k-x}{\Delta}\rho(a+\Delta,\,x-2a)+\binom{x}{a'}\binom{k-x}{\Delta'}\rho(a'+\Delta',\,x-2a')\geq\binom{k-x}{\Delta}\binom{x}{a}\rho(a+\Delta,\,k-x-2\Delta)+\binom{k-x}{\Delta'}\binom{x}{a'}\rho(a'+\Delta',\,k-x-2\Delta'). \tag{7}$$

B2. It is shown that for any two natural numbers a *and* Δ *such that* $\frac{x}{2}\leq a\leq x$*,* $\Delta\leq(k-x)$*,* $\Delta-a<\frac{k-2x}{2}$*, and* $\Delta+a>\frac{k}{2}$*, if* $\rho(a+\Delta,\,x-2a)<\rho(a+\Delta,\,k-x-2\Delta)$*, there exists two natural numbers* a' *and* Δ' *such that*

$$0\leq a'=\Delta-\frac{k-2x}{2}\leq x \text{ and}$$
$$0\leq\Delta'=a+\frac{k-2x}{2}\leq k-x,$$

and Inequality 7 holds.

B3. According to Equation 5, the probabilities

$$\text{PR}_{==}\big(\mathbf{r}_1,\,\mathbf{r}_2,\,x\big) \text{ and}$$
$$\text{PR}_{==}\big(\mathbf{r}_3,\,\mathbf{r}_4,\,k-x\big)$$

can be expanded as the summation of products that are shown in Table 4. For brevity, the constant multiplier

$$\frac{\binom{k}{k-x}}{2^{2k}}$$

has been omitted from both equations.

$\text{PR}_{==}(\mathbf{r}_1, \mathbf{r}_2, x)$				$\text{PR}_{==}(\mathbf{r}_3, \mathbf{r}_4, k-x)$			
a	**Δ**			**a′**	**Δ′**		
0	0	$\binom{x}{0}\binom{k-x}{0}$	$\rho(0,x)$	0	0	$\binom{k-x}{0}\binom{x}{0}$	$\rho(0,k-x)$
0	1	$\binom{x}{0}\binom{k-x}{1}$	$\rho(1,x)$	1	0	$\binom{k-x}{1}\binom{x}{0}$	$\rho(1,k-x-2)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	$k-x-1$	$\binom{x}{0}\binom{k-x}{k-x-1}$	$\rho(k-x-1,x)$	$k-x-1$	0	$\binom{k-x}{k-x-1}\binom{x}{0}$	$\rho(k-x-1,-k+x+2)$
0	$k-x$	$\binom{x}{0}\binom{k-x}{k-x}$	$\rho(k-x,x)$	$k-x$	0	$\binom{k-x}{k-x}\binom{x}{0}$	$\rho(k-x,-k+x)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
x	0	$\binom{x}{x}\binom{k-x}{0}$	$\rho(x,-x)$	0	x	$\binom{k-x}{0}\binom{x}{x}$	$\rho(x,k-x)$
x	1	$\binom{x}{x}\binom{k-x}{1}$	$\rho(x+1,-x)$	1	x	$\binom{k-x}{1}\binom{x}{x}$	$\rho(x+1,k-x-2)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
x	$k-x-1$	$\binom{x}{x}\binom{k-x}{k-x-1}$	$\rho(k-1,-x)$	$k-x-1$	x	$\binom{k-x}{k-x-1}\binom{x}{x}$	$\rho(k-1,-k+x+2)$
x	$k-x$	$\binom{x}{x}\binom{k-x}{k-x}$	$\rho(k,-x)$	$k-x$	x	$\binom{k-x}{k-x}\binom{x}{x}$	$\rho(k,-k+x)$

Table 4: Pairings of products

B4. Each term on the left-hand side of Table 4 can be paired up with a term on the right-hand side such that the summation of products on the left-hand side of Table 4 is always greater than or equal to the summation of products on the right-hand side, thus, proving the base case.

***Inductive Step**: Assuming that Lemma 1 holds for $b \leq \alpha$ where α is a natural number greater than or equal to 1, it needs to be shown that it also holds for $b = \alpha + 1$. The proof sketch is below:*

S1. Let $\mathbf{r}_i[j]$ denote the group of bits in a record utilization vector r_i that have the same hash value j with respect to the hash function f, which is utilized within h_1 (cf., Definition 6). (Assume that within each $\mathbf{r}_i[j]$, the ordering of bits in $\mathbf{r}_i$ is preserved.)

S2. Recall that the Manhattan distance between any two record utilization vectors is the summation of their individual Manhattan distances within each group of bits that share the same hash value with respect to f (cf., Equation 2). Therefore, since $\delta^M(\mathbf{r}_1, \mathbf{r}_2) = x$,

$$\text{if } \delta^M(\mathbf{r}_1[\alpha], \mathbf{r}_2[\alpha]) = a, \text{ then}$$
$$\delta^M(\mathbf{r}_1[0]\cdots\mathbf{r}_1[\alpha-1], \mathbf{r}_2[0]\cdots\mathbf{r}_2[\alpha-1]) = x - a$$

where $\mathbf{r}_i[0]\cdots\mathbf{r}_i[\alpha-1]$ denotes the concatenation of the corresponding bit vectors.

S3. Also note that $h(\mathbf{r}_1) = h(\mathbf{r}_2)$ if and only if $h(\mathbf{r}_1[j]) = h(\mathbf{r}_2[j])$ for all $j \in \mathbb{Z}_{0\cdots\alpha}$. The reason is that $h = h_2 \circ h_1$ and $h(\mathbf{r}_1)$ (respectively, $h(\mathbf{r}_2)$) corresponds to the bit vector that is produced by interleaving the bits in the binary representations of $h_1(\mathbf{r}_1[0])$ $\ldots$ $h_1(\mathbf{r}_1[\alpha])$ (respectively, $h_1(\mathbf{r}_2[0])$ $\ldots$ $h_1(\mathbf{r}_2[\alpha])$) and cutting off the rightmost Υb bits [64]. Consequently, for $h(\mathbf{r}_1) = h(\mathbf{r}_2)$ to be true, for all $j \in \mathbb{Z}_{0\cdots\alpha}$, $h_1(\mathbf{r}_1[j])$ and $h_1(\mathbf{r}_2[j])$ must have the same sequence of bits except for the rightmost Υ, which means that $h(\mathbf{r}_1[j])$ and $h(\mathbf{r}_2[j])$.

S4. As a consequence of statements S2 and S3, the following property holds:

$$\text{PR}_{==}(\mathbf{r}_1, \mathbf{r}_2, x) = \sum_{a=0}^{x} \text{PR}_{==}(\mathbf{r}_1[0]\cdots\mathbf{r}_1[\alpha-1], \mathbf{r}_2[0]\cdots\mathbf{r}_2[\alpha-1], x-a) \times \text{PR}_{==}(\mathbf{r}_1[\alpha], \mathbf{r}_2[\alpha], a) \times \text{PR}_a$$

where $\text{PR}_a \in \{\text{PR}_0, \ldots, \text{PR}_x\}$ denotes a constant that represents the probability that $\delta^M(\mathbf{r}_1[\alpha], \mathbf{r}_2[\alpha]) = a$ among all possible configurations with $\delta^M(\mathbf{r}_1, \mathbf{r}_2) = x$.

S5. Statement S2 holds also for $\mathbf{r}_3$ and $\mathbf{r}_4$; therefore, since $\delta^M(\mathbf{r}_3, \mathbf{r}_4) = \lambda b - x$,

$$\text{if } \delta^M(\mathbf{r}_3[\alpha], \mathbf{r}_4[\alpha]) = a', \text{ then}$$
$$\delta^M(\mathbf{r}_3[0]\cdots\mathbf{r}_3[\alpha-1], \mathbf{r}_4[0]\cdots\mathbf{r}_4[\alpha-1]) = \lambda b - x - a'$$

where $\mathbf{r}_i[0]\cdots\mathbf{r}_i[\alpha-1]$ denotes the concatenation of the corresponding bit vectors.

S6. Since there are $\lambda b - \lambda$ bits in $\mathbf{r}_3[0]\cdots\mathbf{r}_3[\alpha-1]$ and $\mathbf{r}_4[0]\cdots\mathbf{r}_4[\alpha-1]$, the edit distance between these two

bit vectors can be at most $\lambda b-\lambda$, thus,

$$\delta^M(\mathbf{r}_3[0]\cdots\mathbf{r}_3[\alpha-1],\, \mathbf{r}_4[0]\cdots\mathbf{r}_4[\alpha-1]) \leq \lambda b-\lambda.$$

Therefore, $(\lambda-x) \leq a' \leq \lambda$ must hold.

S7. Consequently, similar to S4, the following statement must be true:

$$\text{PR}_{==}\big(\mathbf{r}_3,\, \mathbf{r}_4,\, \lambda b-x\big) = \sum_{a'=\lambda-x}^{\lambda} \text{PR}_{==}\big(\mathbf{r}_3[0]\cdots\mathbf{r}_3[\alpha-1],\, \mathbf{r}_4[0]\cdots\mathbf{r}_4[\alpha-1],\, \lambda b-x-a'\big) \times \text{PR}_{==}\big(\mathbf{r}_3[\alpha],\, \mathbf{r}_4[\alpha],\, a'\big) \times \text{PR}_{a'}$$

where $\text{PR}_{a'} \in \{\text{PR}_{\lambda-x}, \ldots, \text{PR}_{\lambda}\}$ denotes a constant that represents the probability that $\delta^M(\mathbf{r}_3[\alpha],\, \mathbf{r}_4[\alpha]) = a'$ among all possible configurations with $\delta^M(\mathbf{r}_3,\, \mathbf{r}_4) = \lambda b - x$.

S8. The possible configurations in the summations in S4 and S7 for $\text{PR}_{==}\big(\mathbf{r}_1,\, \mathbf{r}_2,\, x\big)$ and $\text{PR}_{==}\big(\mathbf{r}_3,\, \mathbf{r}_4,\, \lambda b-x\big)$ can be paired up such that the product on the left hand side is always greater than or equal to the product on the right hand side, which means:

$$\text{PR}_{==}\big(\mathbf{r}_1,\, \mathbf{r}_2,\, x\big) \geq \text{PR}_{==}\big(\mathbf{r}_3,\, \mathbf{r}_4,\, \lambda b - x\big).$$

Thus, Lemma 6 holds also for $b = \alpha + 1$.

Theorem 3 *Let $\delta^M(\mathbf{r}_i,\, \mathbf{r}_j)$ denote the Manhattan distance between any two record utilization vectors $\mathbf{r}_i$ and $\mathbf{r}_j$ with size λb, and let m denote the number of rightmost bits that are dropped by h_2, which is utilized in h (cf., Definition 1). For any $\delta^M \leq \frac{\lambda}{2}$, h is $(\delta^M,\, \lambda b - \delta^M,\, \text{PR}_1,\, \text{PR}_2)$-sensitive for some $\text{PR}_1 \geq \text{PR}_2$ where $h = h_2 \circ h_1$ (cf., Definition 1), λ is an even and positive number, $b \in \mathbb{Z}_{1\,\ldots\,\infty}$ denotes the number of entries in the record utilization counters produced by h_1, $m = \Upsilon b$, and $\Upsilon \in \mathbb{Z}_{1\,\ldots\,\lambda-1}$.*

Proof. *The proof steps are as follows:*

S1. PR_1 corresponds to the posterior probability that $h(\mathbf{r}_i) = h(\mathbf{r}_j)$ for any two record utilization vectors $\mathbf{r}_i$ and $\mathbf{r}_j$ with size λb such that $\delta^M(\mathbf{r}_i, \mathbf{r}_j) \leq \delta^M$ [73].

S2. PR_2 corresponds to the posterior probability that $h(\mathbf{r}_i) \neq h(\mathbf{r}_j)$ for any two record utilization vectors $\mathbf{r}_i$ and $\mathbf{r}_j$ with size λb such that $\delta^M(\mathbf{r}_i, \mathbf{r}_j) \geq \lambda b - \delta^M$ [73].

S3. Note that

$$\text{PR}_1 = \frac{\sum_{x=0}^{\delta^M} \text{PR}_{==}\big(\mathbf{r}_i,\, \mathbf{r}_j,\, x\big)\, \binom{\lambda b}{\lambda b-x}}{2^{\lambda b}}$$

$$\text{PR}_2 = \frac{\sum_{x=0}^{\delta^M} \text{PR}_{==}\big(\mathbf{r}_i,\, \mathbf{r}_j,\, \lambda b-x\big)\, \binom{\lambda b}{x}}{2^{\lambda b}}.$$

S4. For all $x \in \mathbb{Z}_{0\,\ldots\,\delta^M}$ $\binom{\lambda b}{\lambda b-x} = \binom{\lambda b}{x}$.

S5. Therefore, for all $x \in \mathbb{Z}_{0\,\ldots\,\delta^M}$

$$\frac{\binom{\lambda b}{\lambda b-x}}{2^{\lambda b}} = \frac{\binom{\lambda b}{x}}{2^{\lambda b}}.$$

S6. According to Lemma 1,

$$\text{PR}_{==}\big(\mathbf{r}_i,\, \mathbf{r}_j,\, x\big) \geq \text{PR}_{==}\big(\mathbf{r}_i,\, \mathbf{r}_j,\, \lambda b - x\big).$$

Consequently, $\text{PR}_1 \geq \text{PR}_2$. □

Theorem 3 suggests that h is a function with locality-sensitive properties, and can be used to approximate the clustering problem. However, it must be noted that the sensitivity analysis of h is conservative. In other words, it is believed that stronger statements can be made about h, in particular, due to the empirical observation that $\text{PR}_{==}\big(\mathbf{r}_i,\, \mathbf{r}_j,\, x\big)$ is a monotonically decreasing function. Proving this conjecture is left as future work.

5.2 Achieving and Maintaining Tighter Bounds on Tunable-LSH

Next, we demonstrate how it is possible to reduce the approximation error of h_1. We first define *load factor* of a record utilization counter entry.

Definition 2 (Load Factor) Given a record utilization counter $\mathbf{c} = (c[0], \ldots, c[b-1])$ with size b, the *load factor* of the i^{th} entry is $c[i]$.

Theorem 4 (Effects of Grouping) *Given two record utilization vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ with size k, let $\mathbf{c_1}$ and $\mathbf{c_2}$ denote two record utilization counters with size $b = 1$ such that $\mathbf{c_1} = h_1(\mathbf{r_1})$ and $\mathbf{c_2} = h_1(\mathbf{r_2})$. Then,*

$$\text{Pr}\left(\begin{array}{l|l} \delta^M(\mathbf{c_1}, \mathbf{c_2}) & c_1[0] = l_1 \text{ AND} \\ \quad = \delta^M(\mathbf{r_1}, \mathbf{r_2}) & c_2[0] = l_2 \end{array}\right) = \gamma \tag{8}$$

where

$$\gamma = \frac{\binom{l_{max}}{l_{min}}\binom{k}{l_{max}}}{\binom{k}{l_{max}}\binom{k}{l_{min}}} \tag{9}$$

and

$$l_{max} = max(l_1, l_2)$$
$$l_{min} = min(l_1, l_2).$$

Proof. *Let $\mathbf{r}_{max}$ denote the record utilization vector with the most number of 1-bits among $\mathbf{r_1}$ and $\mathbf{r_2}$, and let $\mathbf{r}_{min}$ denote the vector with the least number of 1-bits. When $b = 1$, $\delta^M(\mathbf{c_1}, \mathbf{c_2}) = \delta(\mathbf{r_1}, \mathbf{r_2})$ holds if and only if the number of 1-bits on which $\mathbf{r_1}$ and $\mathbf{r_2}$ are aligned is l_{min} because in that case, both $\delta^M(\mathbf{c_1}, \mathbf{c_2})$ and $\delta(\mathbf{r_1}, \mathbf{r_2})$*

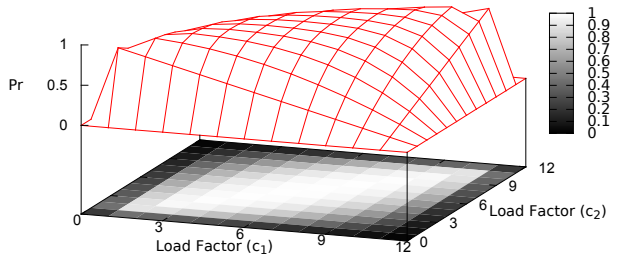


Fig. 3: $\Pr(\delta^M \neq \delta)$ for $k = 12$, $b = 1$ and across varying load factors

are equal to $l_{max} - l_{min}$ *(note that* $\delta^M(\mathbf{c_1}, \mathbf{c_2})$ *is always equal to* $l_{max} - l_{min}$*). Assuming that the positions of* 1*-bits in* $\mathbf{r}_{max}$ *are fixed, there are* $\binom{l_{max}}{l_{min}}$ *possible ways of arranging the* 1*-bits of* $\mathbf{r}_{min}$ *such that* $\delta(\mathbf{r}_1, \mathbf{r}_2) = l_{max} - l_{min}$*. Since the* 1*-bits of* $\mathbf{r}_{max}$ *can be arranged in* $\binom{k}{l_{max}}$ *different ways, there are* $\binom{l_{max}}{l_{min}}\binom{k}{l_{max}}$ *combinations such that* $\delta^M(\mathbf{c}_1, \mathbf{c}_2) = \delta(\mathbf{r}_1, \mathbf{r}_2)$*. Note that in total, the bits of* $\mathbf{r_1}$ *and* $\mathbf{r_2}$ *can be arranged in* $\binom{k}{l_{max}}\binom{k}{l_{min}}$ *possible ways; therefore, Eqns. 8 and 9 describe the posterior probability that* $\delta^M(\mathbf{c_1}, \mathbf{c_2}) = \delta(\mathbf{r_1}, \mathbf{r_2})$*, given* $c_1[0] = l_1$ *and* $c_2[0] = l_2$*.* ■

According to Eqns. 8 and 9 in Thm. 4, the probability that $\delta^M(\mathbf{c_1}, \mathbf{c_2})$ is an approximation of $\delta(\mathbf{r_1}, \mathbf{r_2})$, but that it is not exactly equal to $\delta(\mathbf{r_1}, \mathbf{r_2})$, is lower for load factors that are close or equal to zero and likewise for load factors that are close or equal to $\lceil \frac{k}{b} \rceil$ (cf., Fig. 3). This property suggests that by carefully choosing f, it is possible to achieve even tighter error bounds for h_1. For $b \geq 2$, the probabilities for each group of bits need to be multiplied, which was illustrated in the proof of Lemma 1 earlier in Section 5.1. Therefore, the algorithm for tuning f aims to make sure that the load factors are either low or high for as many of the groups as possible.

Contrast the matrices in Fig 2b and Fig 2c, which contain the same query access vectors, but the columns are grouped in two different ways[3]: (i) in Fig. 2b, the grouping is based on the original sequence of execution, and (ii) in Fig. 2c, queries with similar access patterns are grouped together. Fig. 2d and Fig. 2e represent the corresponding record utilization counters for the record utilization vectors in the matrices in Fig. 2b and Fig. 2c, respectively. Take $\mathbf{r_3}$ and $\mathbf{r_5}$, for instance. Their actual Hamming distance with respect to q_0–q_7 is 8. Now consider the transformed matrices. According to Fig. 2d, the Hamming distance lower bound is 0, whereas according to Fig. 2e, it is 8. Clearly, the bounds in the second representation are closer to the original. The reason is as follows. Even though $\mathbf{r_3}$ and $\mathbf{r_5}$ differ on all the bits for q_0–q_7, when the bits are grouped as in Fig. 2b, the counts alone cannot distinguish the two bit vectors. In contrast, if the counts are computed based on the grouping in Fig. 2c (which clearly places the 1-bits in separate groups), the counts indicate that the two bit vectors are indeed different.

The observations above are inline with Thm. 4. Consequently, we make the following optimization. Instead of randomly choosing a hash function, we construct f such that it maps queries with similar access vectors (i.e., columns in the matrix) to the same hash value. This way, it is possible to obtain record utilization counters with entries that have either very high or very low load factors (cf., Def. 1), thus decreasing the probability of error (cf., Thm. 4).

We develop a technique that efficiently determines groups of queries with similar access patterns and adaptively maintains these groups as the access patterns change. Our approach consists of two parts: (i) to approximate the similarity between any two queries, we rely on the Min-Hash scheme [25], and (ii) to adaptively group similar queries, we develop an incremental version of a multidimensional scaling (MDS) algorithm [62].

Min-Hash offers a quick and efficient way of approximating the similarity, (more specifically, the Jaccard similarity [54]), between two sets of integers. Therefore, to use it, the query access vectors in our conceptualization need to be translated into a set of positional identifiers that correspond to the records for which the bits in the vector are set to 1.[4] For example, according to Fig. 2a, $\mathbf{q_1}$ should be represented with the set $\{0, 5, 6\}$ because r_0, r_5 and r_6 are the only records for which the bits are set to 1. Note that we do not need to store the original query access vectors at all. In fact, after the access patterns over a query are determined, we compute and store only its Min-Hash value. This is important for keeping the memory overhead of our algorithm low.

Queries with similar access patterns are grouped together using a multidimensional scaling (MDS) algorithm [58] that was originally developed for data visualization, and has recently been used for clustering [23]. Given a set of points and a distance function, MDS assigns coordinates to points such that their original distances are preserved as much as possible. In one efficient implementation [62], each point is initially assigned a random set of coordinates, but these coordinates are adjusted iteratively based on a spring-force analogy. That is, it is assumed that points exert a force on each other that is proportional to the difference between their actual and observed distances, where the

[3] Groups are separated by vertical dashed lines.

[4] In practice, this translation is not required because the system maintains positional vectors instead.

Algorithm 4 Reconfigure-F

Require:
$\mathbf{q_t}$: query access vector produced at time t
Ensure:
Coordinates of MDS points are updated, which are used in determining the outcome of f

```
1:  procedure RECONFIGURE-F(q_t)
2:      pos ← (begin + size) % k
3:      S[pos].clear()
4:      N[pos].clear()
5:      X[pos] ← −0.5 + rand() / RAND-MAX
6:      V[pos] ← 0
7:      H[pos] ← MIN-HASH(q_t)
8:      if size < k then
9:          size += 1
10:     else
11:         begin = (begin + 1) % k
12:     end if
13:     for i ← 0, i < size, i++ do
14:         x ← (begin + i) % k
15:         UPDATE-S-AND-N(x)
16:         UPDATE-VELOCITY(x)
17:     end for
18:     for i ← 0, i < size, i++ do
19:         x ← (begin + i) % k
20:         UPDATE-COORDINATES(x)
21:     end for
22: end procedure
```

latter refers to the distance that is computed from the algorithm-assigned coordinates. These forces are used for computing the current velocity (V in Table 2) and the approximated coordinates of a point (X in Table 2). The intuition is that, after successive iterations, the system will reach equilibrium, at which point the approximated coordinates can be reported. Since computing all pairwise distances can be prohibitively expensive, the algorithm relies on a combination of sampling ($S[]$ in Table 2) and maintaining a list of each point's nearest neighbours ($N[]$ in Table 2)—only these distances are used in computing the net force acting on a point. Then, the nearest neighbours are updated in each iteration by removing the most distant neighbour of a point and replacing it with a new point from the random sample if the distance between the point and the random sample is smaller than the distance between the point and its most distant neighbour.

There are multiple reasons for our choice of MDS over other clustering approaches such as single-linkage clustering or k-means [55] (even though Tunable-LSH is agnostic to this choice). First of all, MDS is less sensitive to the shape of the underlying clusters (e.g., as opposed to single-linkage) and the initial choice of clusters (e.g., as opposed to k-means) largely due to the reliance on random sampling (i.e., $S[]$). Second, it allows the trade-off between computational overhead vs. quality of clustering to be tuned more precisely by controlling multiple parameters such as (i) the size of vectors $S[]$ and $N[]$ (independently), (ii) the number of iterations, and (iii) the number of output dimensions. Nevertheless, this algorithm cannot be used directly for our purposes because it is not incremental. Therefore, we propose a revised MDS algorithm that incorporates the following modifications:

1. In our case, each point in the algorithm represents a query access vector. However, since we are not interested in visualizing these points, but rather clustering them, we configure the algorithm to place these points along a single dimension. Then, by dividing the coordinate space into consecutive regions, we are able to determine similar query access vectors.
2. Instead of computing the coordinates of all of the points at once, our version makes incremental adjustments to the coordinates every time reconfiguration is needed.

The revised algorithm is given in Algorithm 4. First, the algorithm decides which MDS point to assign to the new query access vector $\mathbf{q_t}$ (line 2). It clears the array and the heap data structures containing, respectively, (i) the randomly sampled, and (ii) the neighbouring set of points (lines 3–4). Furthermore, it assigns a random coordinate to the point within the interval $[-0.5, 0.5]$ (line 5), and resets its velocity to 0 (line 6). Next, it computes the Min-Hash value of $\mathbf{q_t}$ and stores it in $H[\text{pos}]$ (line 7). Then, it makes two passes over all the points in the system (lines 13–21), while first updating their sample and neighbouring lists (line 15), computing the net forces acting on them based on the Min-Hash distances and updating their velocities (line 16); and then updating their coordinates (line 20).

The procedures used in the last part are implemented in a similar way as the original algorithm [62]; that is, in line 15, the sampled points are updated, in line 16, the velocities assigned to the MDS points are updated, and in line 20, the coordinates of the MDS points are updated based on these updated velocities. However, our implementation of the Update-Velocity procedure (line 16) is slightly different than the original. In particular, in updating the velocities, we use a decay function so that the algorithm forgets "old" forces that might have originated from the elements in $S[]$ and $N[]$ that have been assigned to new query access vectors in the meantime. Note that unless one keeps track of the history of all the forces that have

Algorithm 5 Hash Function f

Require:
t: sequence number of a query access vector
Ensure:
$f(t)$ is computed and returned
1: **procedure** F(t)
2: pos $\leftarrow t \,\%\, k$
3: (lo, hi) $\leftarrow$ GROUP-BOUNDS(X[pos])
4: coid $\leftarrow$ CENTROID(lo, hi)
5: **return** HASH(coid) $\%\, b$
6: **end procedure**

acted on every point in the system, there is no other way of "undoing" or "forgetting" these "old" forces.

Given the sequence number of a query access vector (t), the outcome of the hash function f is determined based on the coordinates of the MDS point that had previously been assigned to the query access vector by the RECONFIGURE procedure. To this end, the k points are sorted based on their coordinates, and the coordinate space is divided into b groups containing points with consecutive coordinates such that there are at most $\lceil \frac{k}{b} \rceil$ points in each group. Then, one option is to use the group identifier, which is a number in $\mathbb{Z}_{0 \cdots b-1}$, as the outcome of f, but there is a problem with this naïve implementation. Specifically, we observed that even though the *relative* coordinates of MDS points within the "same" group may not change significantly across successive calls to the RECONFIGURE procedure, points within a group, as a whole, may shift. This is an inherent (and in fact, a desirable) property of the incremental algorithm. However, the problem is that there may be far too many cases where the group identifier of a point changes just because the absolute coordinates of the group have changed, even though the point continues to be part of the "same" group. To solve this problem, we rely on a method of computing the centroid within a group by taking the MIN-HASH of the identifiers of points within that group such that these centroids rarely change across successive iterations. Then, we rely on the identifier of the centroid, as opposed to its coordinates, to compute the group number, hence, the outcome of f. The pseudocode of this procedure is given in Algorithm 5.

We make one last observation. Internally, MIN-HASH uses multiple hash functions to approximate the degree to which two sets are similar [25]. It is also known that increasing the number of internal hash functions used (within MIN-HASH) should increase the overall accuracy of the MIN-HASH scheme. However, as unintuitive as it may seem, in our approach, we use only a single hash function within MIN-HASH, yet, we are still able to achieve sufficiently high accuracy. The reason is as follows. Recall that Algorithm 4 relies on multiple pairwise distances to position every point. Consequently, even though individual pairwise distances may be inaccurate (because we are just using a single hash function within MIN-HASH), collectively the errors are cancelled out, and points can be positioned accurately on the MDS coordinate space.

	0	**1**	**2**	**3**	**4**	**5**
$t = 0$	□	□	□	∅		
$t = \lceil k/3 \rceil$		□	□	□	∅	
$t = \lceil 2k/3 \rceil$			□	□	□	∅
$t = k$	∅			□	□	□
$t = \lceil 4k/3 \rceil$	□	∅			□	□
$t = \lceil 5k/3 \rceil$	□	□	∅			□

Fig. 4: Assuming $b = 3$, □ indicates the allowed locations at each time tick, and ∅ indicates the counter to be reset.

5.3 Resetting Old Entries in Record Utilization Counters

Once the group identifier is computed (cf., Algorithm 5), it should be straightforward to update the record utilization counters (cf., line 8 in Algorithm 2). However, unless we maintain the original query access vectors, we have no way of knowing which counters to decrement when a query access vector becomes stale, as maintaining these original query access vectors is prohibitively expensive. Therefore, we develop a more efficient scheme in which old values can also be removed from the record utilization counters.

Instead of maintaining b entries in every record utilization counter, we maintain twice as many entries ($2b$). Then, whenever the TUNE procedure is called, instead of directly using the outcome of $f(t)$ to locate the counters to be incremented, we map $f(t)$ to a location within an "allowed" region of consecutive entries in the record utilization counter (cf., line 8 in Algorithm 2). At every $\lceil \frac{k}{b} \rceil^{\text{th}}$ iteration, this allowed region is shifted by one to the right, wrapping back to the beginning if necessary. Consider Fig. 4. Assuming that $b = 3$ and that at time $t = 0$ the allowed region spans entries from 0 to $(b-1)$, at time $t = \lceil \frac{k}{b} \rceil$, the region will span entries from 1 to b; at time $t = k$, the region will span entries from b to $2b - 1$; and at time $t = \lceil \frac{4k}{b} \rceil$, the region will span entries 0 and those from $b + 1$ to $2b - 1$.

Since $f(t)$ produces a value between 0 and $b-1$ (inclusive), whereas the entries are numbered from 0 to $2b - 1$ (inclusive), the RECONFIGURE procedure in Al-

gorithm 2 uses $f(t)$ as follows. If the outcome of $f(t)$ directly corresponds to a location in the allowed region, then it is used. Otherwise, the output is incremented by b (cf., line 8 in Algorithm 2). Whenever the allowed region is shifted to the right, it may land on an already incremented entry. If that is the case, that entry is reset, thereby allowing "old" values to be forgotten (cf., line 11 in Algorithm 2). These are shown by $\emptyset$ in Fig. 4. This scheme guarantees any query access pattern that is less than k steps old is remembered, while any query access pattern that is more than $2k$ old is forgotten.

5.4 Deletions and Insertions

Currently, Tunable-LSH does not support the scenario where more triples are added to the database (e.g., the streaming RDF case or simply insertions). However, extending Tunable-LSH to support this use case is possible. First, every new insertion can be treated as a new record utilization vector with a corresponding record utilization counter of all zeros. Such records are initially unclustered (or they go into a designated cluster). As queries are executed and query access vectors start including the newly added records, the counters can be updated (as is done for any other record), which would result in the final hash values to be updated as well. The one challenge is that Min-Hash, which is an optimization used by Tunable-LSH to cluster the query access vectors, will become less sensitive as query access vectors grow in size. The loss in optimization can be compensated by periodically adjusting the Min-Hash function, e.g., simply by increasing the number of hash functions that are internally used for constructing Min-Hash. With respect to deletion of records, there are two options: either rely on the inherent timeout mechanism of Tunable-LSH to gradually reset old entries in record utilization counters or explicitly set the counters to zero and move the records to the designated clusters. We treat these ideas as future work.

6 Experimental Evaluation

In this section, we evaluate Tunable-LSH in three sets of experiments. First, we evaluate it within chameleon-db, a prototype RDF data management system. Second, we evaluate it within a hashtable implementation. Third, we evaluate Tunable-LSH in isolation, to understand how it behaves under different types of scenarios.

6.1 Effectiveness of Tunable-LSH

In our experiments, we study the effectiveness of Tunable-LSH by implementing it in a prototype RDF data management system called chameleon-db. We chose chameleon-db as the prototype engine because it does not have a fixed physical layout and it allows us to experiment with different types of low-level physical layouts. While the details of chameleon-db have been discussed elsewhere [10, 14, 15], for completeness, we provide a summary next.

chameleon-db supports the Basic Graph Pattern subfragment of SPARQL [45] and currently does not support updates. In chameleon-db, RDF is conceptually represented as a graph, and the graph is partitioned into very small chunks called *group-by-query* (GbyQ) *clusters*. While there is no restriction on the size of a GbyQ cluster, chameleon-db operates with very small clusters, each containing at most tens of edges. On disk, each GbyQ cluster is serialized as a sequence of RDF triples that make up that cluster. When a cluster is brought into the buffer pool, the buffer manager automatically converts the serialized representation into an adjacency list representation in memory. Unlike conventional systems, chameleon-db relies on partially-built, workload-adaptive indexes similar to database cracking [51, 52]. This adaptive indexing allows the underlying GbyQ clustering to be updated efficiently. Thus, the storage advisor of chameleon-db triggers periodically to compute a new clustering based on the most recently executed queries in the workload and updates the underlying physical representation. In our experiments, we manually trigger the tuning advisor. The storage advisor of chameleon-db computes the RDF triples (or equivalently, the vertices and edges) to be clustered together [into GbyQ clusters] using either the conventional clustering algorithm discussed in [15] or Tunable-LSH. The query access vectors, which constitute the input to these algorithms, are constructed from the query results alone.

The following experiments are performed on a commodity machine with Intel® Core™ i5-8400 2.80 GHz CPU (x86_64), 4×16 GB Corsair Vengeance 2400 MHz RAM, and a Seagate ST2000DM006-2DM1 2 TB HDD with more than 500 GB of free space. The operating system is Ubuntu 16.04.5 LTS.

In these first set of experiments, we evaluate Tunable-LSH within chameleon-db. We use the Waterloo SPARQL Diversity Test Suite (WatDiv) [12] because WatDiv can generate datasets with skew and queries that are more diverse than existing benchmarks (it has already been demonstrated that there could be up to three orders of magnitude difference between the fastest

and slowest system executing WatDiv's stress testing workloads – even for simple queries) [12]. This includes queries that are very selective as well as those that are not selective at all (i.e., those returning a large chunk of triples in the database), queries where the triple patterns that make up the query are not selective but the query itself is very selective, queries that are composed of only a few triple patterns as well as those that are composed of many triple patterns and so on.

We use the WatDiv *data generator* to create a dataset with 1 billion RDF triples (abbreviated as WatDiv 1B). Then, using the WatDiv *query template generator*, 70 query templates are generated where the largest query consists of 10 triple patterns. Of these 70 templates, 51 are simple (i.e., with less than or equal to 4 triple patterns), 13 are star-shaped (i.e., with greater than 4 triple patterns that are incident on the same vertex), and 6 are either linear-shaped or complex.

In the first experiment, we instantiate each query template with 500 queries and execute the first 100 queries in chameleon-db using its baseline, triple-based partitioning (abbreviated Triple), and then tune the system with the conventional clustering algorithm described earlier (abbreviated Clustered). We repeat this same experiment where chameleon-db is tuned instead with Tunable-LSH (abbreviated TLSH).

MEAN (ms)	Triple	TLSH	Clustered
Simple	1,283.49	57.70	24.68
Star	648.37	484.86	1.60
Linear/Complex	3,525.66	660.19	790.96
ALL	1,349.52	187.88	85.33

Table 5: Comparison of Tunable-LSH to conventional clustering in chameleon-db on WatDiv 1B

Table 5 reports the average query execution times for each of the three approaches mentioned above, aggregated across the different query types in the workload (i.e., Simple, Star, Linear/Complex). For the Clustered and TLSH approaches, the time to compute the new clustering of triples as well as physically updating the layout are included in the reported averages.

There are reasons why query execution in chameleon-db is slower with Tunable-LSH than the conventional approach: Tunable-LSH is an approximate algorithm by design; therefore, the clusters generated by Tunable-LSH are not as accurate as the ones that are generated by the conventional approach, which is the expected behavior. Second, we note that Tunable-LSH and the conventional clustering algorithm generate large clusters for star-shaped queries because the query results share lots of common vertices and edges, but Tunable-LSH breaks down the clusters into page-sized chunks (4KB in this case) whereas the conventional algorithm favors larger clusters that are not broken down. The former layout leads to more frequent query decomposition, which chameleon-db is not designed to handle efficiently [15], and explains the larger gap for star-shaped queries. Improving chameleon-db's query optimizer is beyond the scope of this paper.

While the query execution times with Tunable-LSH are slightly slower, in this experiment, we have observed that the computational overhead of conventional clustering was an order of magnitude larger than Tunable-LSH (1070 vs 26 milliseconds). This gap increases even more in subsequent experiments, where it is evident that the conventional approach does not scale well to query mixes and/or changing workloads (i.e., with respect to memory consumption and computational overhead).

We repeat the first experiment also on a crawl of the DBpedia dataset containing 50 million triples [63]. Since chameleon-db supports only basic graph patterns (BGPs), the query logs provided by the benchmark were utilized to extract 14 BGP templates[5]. Note that most queries in the DBpedia query logs are repetitive and a large portion of the queries consist of only a single or few triple patterns [12]. To avoid bias, for queries having 3–10 triple patterns, the most frequent templates were selected and then each template was instantiated with 25 queries, making sure that there is at least one instantiation (per template) that returns a non-empty result.

Our evaluations on the DBpedia dataset are consistent with those on WatDiv (Table 6). For Q6, analysis of the query logs points to problems with chameleon-db's query optimizer as opposed to Tunable-LSH (as is the case for the expensive WatDiv queries). In contrast to the WatDiv experiment, Tunable-LSH in the DBpedia experiment is able to achieve a clustering quality that is almost as good as the conventional algorithm, which is reflected in the query execution times. Lastly, as before, Tunable-LSH is able to complete clustering in milliseconds on average, whereas the conventional clustering algorithm takes more than 10 minutes.

In the second experiment, we compare chameleon-db tuned with Tunable-LSH against the latest versions of four modern, open-source RDF data management systems: QLever (June 6, 2018) [19], gStore v0.6.0 (April 25, 2018) [77,79], gh-rdf3x (August 16, 2013) [66], and TripleBit (September 19, 2014) [76]. QLever is an RDF engine that supports SPARQL and text search;

[5] `https://cs.uwaterloo.ca/~galuc/files/dbpedia-test-queries.tar.gz`

MEAN (ms)	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}
CDB [Clustered]	24.61	6.09	2.96	3.37	5.93	NA	438.39	7.29	NA	5.63	6.84	6.81	16.08	7.46
CDB [TLSH]	27.63	2.64	2.89	3.15	6.41	95957.38	761.38	6.89	41.84	5.63	6.87	6.76	22.34	7.46

Table 6: Comparison of Tunable-LSH to conventional clustering in chameleon-db on DBpedia 50M

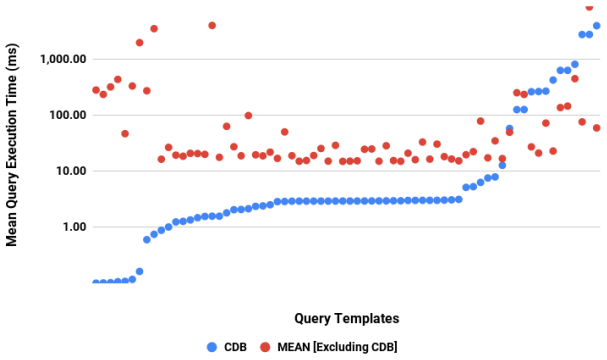


Fig. 5: Performance evaluation of chameleon-db using Tunable-LSH

gStore is a graph-based RDF engine; gh-rdf3x is a more recent and optimized version of the original RDF-3x engine; and TripleBit is an RDF engine that relies on bitmaps for efficient pruning and processing of triples. In this experiment, we use the same workload used in the first experiment and tune chameleon-db independently for each query template. Since chameleon-db is a non-distributed system, for a fair comparison, we also setup the other systems to work on a single-node. Furthermore, we configure these systems so that they make as much use of the available RAM as possible.

Fig. 5 illustrates that using Tunable-LSH, chameleon-db can be tuned to perform well in comparison to other RDF data management systems across a wide range of queries. In particular, for about one third of the query templates, TLSH is two orders of magnitude faster than the average of other systems for the same queries (i.e., the left-hand side of Fig. 5); for about one third of the query templates, TLSH is an order of magnitude faster (i.e., the middle part of Fig. 5); and in the worst case, TLSH achieves performance comparable to other systems (i.e., the right-hand side of Fig. 5).

Table 7 reports the average query execution times of all systems across the 70 query templates (70×500 queries). When executing these workloads, QLever unexpectedly aborted during almost half of the queries. Consequently, while we report QLever's average over the remaining queries, we caution the reader to not read too much into values for QLever. Excluding QLever, chameleon-db tuned with Tunable-LSH is the second fastest system.

In the third experiment, we demonstrate that Tunable-LSH can be used to efficiently tune for workloads that consist of instantiations of *multiple* query templates, which is more realistic. For this experiment, we use the WatDiv stress testing tools to randomly generate workloads consisting of 5, 10 and 15 query templates, respectively, where each workload consists of approximately 10,000 queries in total. In these workloads, approximately 2-in-5 queries are simple, 2-in-5 are star-shaped and 1-in-5 are linear or complex. The underlying dataset is the same as the one used in the previous experiments (i.e., WatDiv 1B). For each workload, we let chameleon-db execute the first 25–35 percent of the workload using its baseline partitioning and then trigger the tuning advisor at a random point in time to dynamically adapt to the workload using Tunable-LSH. The same workloads are also executed on the two fastest systems reported in the previous experiment, namely, gh-rdf3x and TripleBit.

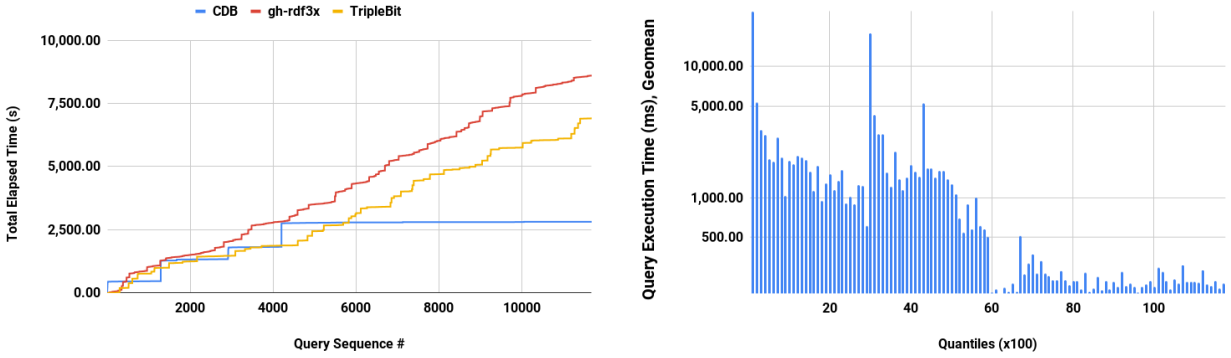
Fig. 6a shows the total elapsed time (in seconds) for all three systems as each system executes the workload that consists of the 10 query template mix. In this workload, after executing the 2912th query, chameleon-db's tuning advisor kicks in, and it starts computing the new partitioning using Tunable-LSH (which takes only 84.4 milliseconds), and the subsequent queries pay the price for physical re-partitioning. As more and more queries are executed and the partitioning gets better, the system speeds up to the extent that it becomes much faster than the two other systems. In fact, just before the execution of the 6000th query, chameleon-db breaks even, and from that point on, it becomes the fastest system (with respect to the end-to-end workload execution time).

Fig. 6b breaks down the total elapsed time for chameleon-db (for the same workload in Fig. 6a) into quantiles of 100 queries and shows the geometric mean time to execute a query within each quantile (note the log-scale). We make multiple observations:

– For quantiles 1–30 during which chameleon-db uses the baseline partitioning, queries gradually become faster. This is due to the adaptive indexing property of chameleon-db: every time chameleon-db encounters a query with a different structure, it cracks down its indexes in an adaptive fashion similar to database cracking [51,52]. In other words, chameleon-

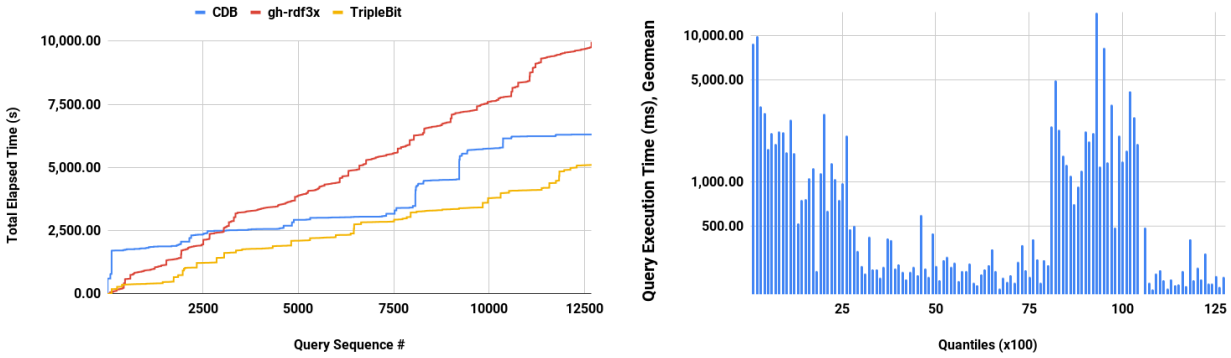
MEAN (ms)	QLever(*)	gStore	gh-rdf3x	TripleBit	CDB [TLSH]
ALL	34.08	844.50	280.09	129.11	187.88

Table 7: Comparison of chameleon-db using TUNABLE-LSH against modern RDF engines



(a) (b)

Fig. 6: Adaptivity of chameleon-db on workloads at 10-template query mix [Triples to TLSH]



(a) (b)

Fig. 7: Adaptivity of chameleon-db on changing workloads at 10-template query mix [TLSH to TLSH]

db pays the price of indexing as queries are executed.

– Between the 30th and 60th quantiles, query execution becomes slower to begin with but then gradually improves again. There are two reasons for this behavior: First, physical re-partitioning incurs I/O, whose price is paid by the queries. Second, as new partitions are created and old partitions are removed, chameleon-db needs to adaptively update its indexes, which also adds overhead.
– After the 60th quantile, the partitioning is optimized and the indexes have fully adapted to the query structures in the workload. From this point on, queries are fast with occasional fluctuations due to the inherent randomness in the workload.

Apart from slightly different break-even points, we verified a very similar pattern with 5 and 15 template query mixes so we did not include these charts in the paper.

In the fourth experiment, we show that TUNABLE-LSH can be used to efficiently tune chameleon-db for changing workloads. In this experiment, the WatDiv stress testing tools have been utilized to randomly generate six sets of seeds consisting of 5, 5, 10, 10, 15 and 15 query templates, respectively. Using each set of seeds, 6 different workloads have been instantiated in total, where each workload contains approximately 6 thousand queries (sometimes slightly less and sometimes slightly more). In these workloads, approximately 2-in-5 queries are simple, 2-in-5 are star-shaped and 1-in-5 are

linear or complex. These workloads have been classified into three groups based on the number of query templates from which each workload was generated (e.g., 5 template workloads, 10 template workloads, etc.). In the experiment, for each workload group, we let chameleon-db execute the first workload using a partitioning that was already computed for the workload using Tunable-LSH (stage-I), switch to the second workload and let the system execute part of the second workload using a sub-optimal partitioning (stage-II), and finally tune the system again for the most recent workload using Tunable-LSH and continue executing the remainder of the second workload (stage-III). The same workloads are also executed on the two fastest systems reported in the previous experiments, namely, gh-rdf3x and TripleBit.

Fig 7a shows the total elapsed time (in seconds) for all three systems for the 10 query template workloads. Fig 7b breaks down the total elapsed time for chameleon-db into quantiles of 100 queries. For the workload portrayed in these figures, stage-I corresponds to queries 1–8064, stage-II to queries 8065–9211 and stage-III to the remaining 9212–12679.

Based on these results, the following observations can be made: First, for the same reasons discussed in the third experiment, query execution times gradually improve during stage-I. Second, during stage-II, we get sub-optimal query performance, which is expected. Third, at the beginning of stage-III, we get even worse performance (due to physical re-clustering and re-learning of the indexes). Last, for the last two thirds of stage-III (once the system has fully adapted), the queries are executed very efficiently.

We tried repeating the third and fourth experiments using the conventional clustering algorithm; however, either we ran out of memory or the algorithm took more than 10 minutes to execute (in comparison to an average of 276ms for Tunable-LSH) and thus, we had to abort the process. Recall that for the first and second experiments, we were able to successfully go through the tuning process but in that case the workloads consisted of single query templates each. This is due to the fact that compared against a conventional clustering algorithm, which can be quadratic in computational complexity, Tunable-LSH, by design, trades off accuracy for execution time. This finding strengthens the motivation to use Tunable-LSH albeit the slightly worse query execution times reported in the first two experiments.

In summary, Tunable-LSH

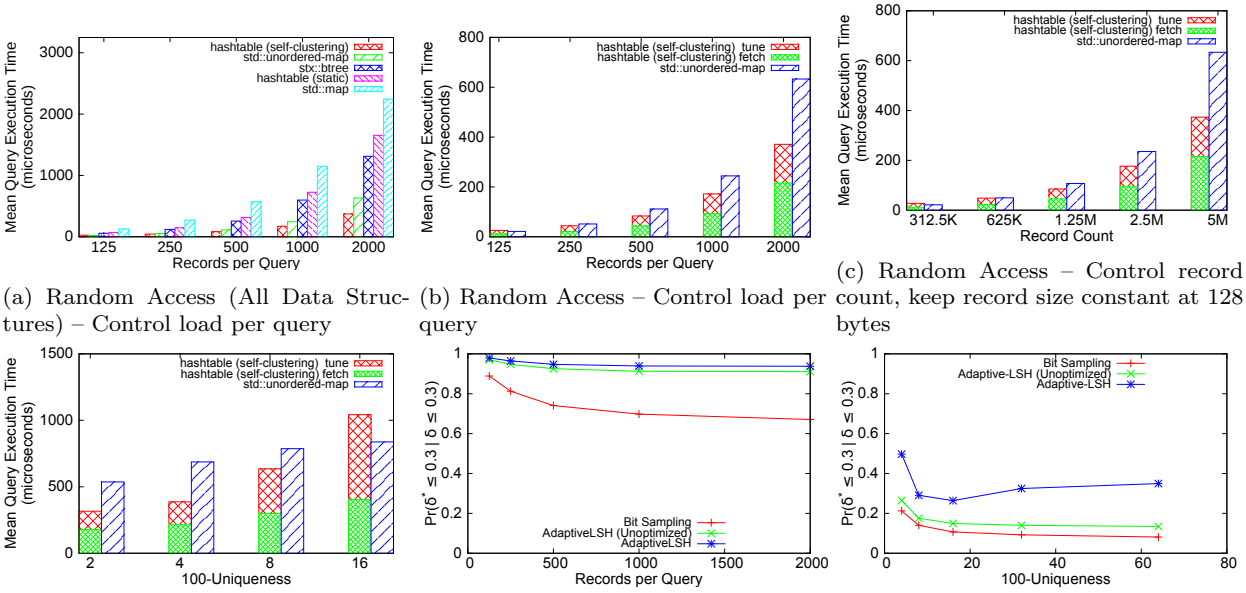
- can be used to tune the physical design of an RDF data management system to yield query execution times that are comparable to a conventional clustering algorithm;
- has a significantly lower computational overhead than conventional clustering, and thus, is a better choice for online tuning;
- can be utilized to tune across a diverse selection of workloads so that it performs comparably with modern open-source RDF data management systems; and
- can be used effectively for tuning for workloads with mixed query types as well as changing workloads.

6.2 Self-Clustering Hashtable

The second experiment evaluates an in-memory hashtable that we developed that uses Tunable-LSH to dynamically cluster records in the hashtable. Hashtables are commonly used in RDF data management systems. For example, the dictionary in an RDF data management system, which maps integer identifiers to URIs or literals (and vice versa), is often implemented as a hashtable [4, 33, 75]. Secondary indexes can also be implemented as hashtables, whereby the hashtable acts as a key-value store and maps tuple identifiers to the content of the tuples. In fact, in chameleon-db, all the indexes are secondary (dense) indexes because instead of relying on any sort order inherent in the data, RDF triples are ordered purely based on the workload.

The hashtable interface is very similar to that of a standard one except that users can optionally mark the beginning and end of queries (i.e., in case they want the records obtained through a sequence of lookups to be clustered together). This information is used to dynamically cluster records such that those that are co-accessed across similar sets of queries also become physically co-located. All of the clustering and re-clustering is transparent to the user, hence, the name, *self-clustering hashtable*.

The self-clustering hashtable has the following advantages and disadvantages: compared to a standard hashtable that tries to avoid hash-collisions, it strives to co-locate records that are accessed together. If the workloads favour a scenario in which many records are frequently accessed together, then we can expect the self-clustering hashtable to have improved fetch times due to better CPU cache utilization, prefetching, etc. [7]. On the other hand, these optimizations come with three types of overhead. First, every time a query is executed, Tunable-LSH needs to be updated (cf., Algorithms 2 and 4). Second, compared to a standard hashtable in which the physical address of a record is determined solely using the underlying hash function (which is deterministic throughout the entire workload), in our case



(a) Random Access (All Data Structures) – Control load per query

(b) Random Access – Control load per query

(c) Random Access – Control record count, keep record size constant at 128 bytes

(d) Random Access – Control workload dynamism

(e) Sensitivity analysis of Tunable-LSH– Control load per query

(f) Sensitivity analysis of Tunable-LSH– Control workload dynamism

Fig. 8: Experimental evaluation of Tunable-LSH in a self-clustering hashtable and the sensitivity analysis of Tunable-LSH

the physical address of a record needs to be maintained dynamically because the underlying hash function is not deterministic (i.e., it is also changing dynamically throughout the workload). Consequently, there is the overhead of going to a lookup table and retrieving the physical address of a record. Third, physically moving records around in the storage system takes time—in fact, this is often an expensive operation. Therefore, the objective of this set of experiments is twofold: (i) to evaluate the circumstances under which the self-clustering hashtable outperforms other popular data structures, and (ii) to understand when the tuning overhead may become a bottleneck. Consequently, we report the end-to-end query execution times, and if necessary, break it down into the time to (i) *fetch* the records, and (ii) *tune* the data structures (which includes all types of overhead listed above).

In our experiments, we compare the self-clustering hashtable to popular implementations of three data structures. Specifically, we use: (i) *std::unordered_map* [3], which is the C++ standard library implementation of a hashtable, (ii) *std::map* [2], which is the C++ standard library implementation of a red-black tree, and (iii) *stx::btree* [22], which is an open source in-memory B+ tree implementation. As a baseline, we also include a static version of our hashtable, i.e., one that does not rely on Tunable-LSH.

We consider two types of workloads: one in which records are accessed *sequentially* (i.e., based on their physical ordering in the storage system) and the other in which records are accessed *randomly*. Each workload consists of 3000 synthetically generated queries (where each query consists of a collection of individual lookups on the hashtable). For each data structure, we measure the end-to-end workload execution time and compute the mean time by dividing the total workload execution time by the number of queries in the workload.

Queries in these workloads consist of changing query access patterns, and in different experiments, we control different parameters such as the number of records that are accessed by queries on average, the rate at which the query access patterns change in the workload, etc. We repeat each experiment 20 times over workloads that are randomly generated with the same characteristics (e.g., average number of records accessed by each query, how fast the workload changes, etc.) and report averages across these 20 runs. We do not report standard errors as they are negligibly small.

For the sequential case, *stx::btree* and *std::map* outperform the hashtables, which is expected because once the first few records are fetched from main-memory, the remaining ones can already be prefetched into the CPU cache (due to the predictability of the sequential access pattern). Therefore, for the remaining part, we focus on the random access scenario, which can be a bottleneck even in systems like RDF-3x [66] that have clustered indexes over all permutations of attributes. For a detailed explanation, we refer the reader to [13].

In this experiment, we control the number of records that a query needs to access (on average), where each record is 128 bytes. Fig. 8a compares all the data structures with respect to their end-to-end (mean) query execution times. Three observations stand out: first, in the random access case, the self-clustering hashtable as well as the standard hashtable perform much better than the other data structures, which is what would be expected. This observation holds also for the subsequent experiments, therefore, for presentation purposes, we do not include these data structures in Fig. 8b–8d. Second, the baseline static version of our hashtable (i.e., without Tunable-LSH) performs much worse than the standard hashtable, even worse than a B+ tree. This suggests that our implementation can be optimized further, which might improve the performance of the self-clustering hashtable as well (this is left as future work). Third, as the number of records that a query needs to access increases, the self-clustering hashtable outperforms all the other data structures, which verifies our initial hypothesis.

For the same experiment above, Fig. 8b focuses on the self-clustering hashtable versus the standard hashtable, and illustrates why the performance improvement is higher (for the self-clustering hashtable) for workloads in which queries access more records. Note that while the *fetch* time of the self-clustering hashtable scales proportionally with respect to std::unordered_map, the *tune* overhead is proportionally much lower for workloads in which queries access more records. This is because with increasing records per query count, records can be re-located in batches across the pages in main-memory as opposed to moving individual records around.

Next, we keep the average number of records that a query needs to access constant at 2000, but control the number of records in the database. As in the previous experiment, each record is 128 bytes. As illustrated in Fig. 8c, increasing the number of records in the database (i.e., scaling-up) favours the self-clustering hashtable. The reason is that when there are only a few records in the database, the records are likely clustered to begin with. We repeat the same experiment, but this time, by controlling the record size and keeping the database size constant at 640 megabytes. Surprisingly, the relative improvement with respect to the standard hashtable remains more or less constant, which indicates that the improvement is largely dominated by the size of the database, and increasing it is to the advantage of the self-clustering hashtable.

Finally, we evaluate how sensitive the self-clustering hashtable is to the dynamism in the workloads. Note that for the self-clustering hashtable to be useful at all, the workloads need to be somewhat predictable. That is, if records are physically clustered but are never accessed in the future, then the clustering efforts are wasted. To verify this hypothesis, we control the expected number of query clusters (i.e., queries with similar but not exactly the same access vectors) in any 100 consecutive queries in the workloads that we generate. Let us call this property of the workload its 100-*Uniqueness*. Fig. 8d illustrates how the *tuning* overhead can start to become a bottleneck as the workloads become more and more dynamic, to the extent of being completely unique, i.e., each query in the workload accesses a distinct set of records.

6.3 Sensitivity Analysis of Tunable-LSH

In the final set of experiments, we evaluate the sensitivity of Tunable-LSH in isolation, that is, without worrying about how it affects physical clustering, and compare it to three other hash functions: (i) a standard non-locality sensitive hash function [1], (ii) bit-sampling, which is known to be locality-sensitive for Hamming distances [53], and (iii) Tunable-LSH without the optimizations discussed in Section 5. These comparisons are made across workloads with different characteristics (i.e., dense vs. sparse, dynamic vs. stable, etc.) where parameters such as the average number of records accessed per query and the expected number of query clusters within any 100-consecutive sequence of queries in the workload are controlled.

Our evaluations indicate that Tunable-LSH generally outperforms its alternatives. Due to space considerations, we summarize our most important observations.

Fig. 8e shows how the probability that *the evaluated hash functions place records with similar utilization vectors to nearby hash values* changes as the average number of records that each query accesses is increased. In computing these probabilities, both the original distances (i.e., δ) and the distances over the hashed values (i.e., δ^*) are normalized with respect to the maximum distance in each geometry. It can be observed that both versions of Tunable-LSH are better than bit-sampling especially when the number of records in a query increases.

Fig. 8f shows how the quality of clustering changes as the workloads become more and more dynamic. As illustrated in Fig. 8f, Tunable-LSH achieves higher probability even when the workloads are dynamic. Note that the unoptimized version of Tunable-LSH behaves no worse than a static locality-sensitive hash function, such as bit sampling, which conforms to the theorems in Section 5.1. We have not included the results

on the standard non-locality sensitive hash function, because, as one might guess, it has a probability distribution that is completely unaligned with our clustering objectives.

7 Conclusions and Future Work

In this paper, we introduce Tunable-LSH, which is a locality-sensitive hashing scheme, and demonstrate its use in clustering records in an RDF data management system. In particular, we keep track of records that are accessed by the same query but are fragmented across the pages in the database and use Tunable-LSH to decide, in constant-time, where a record needs to be placed in the storage system. Tunable-LSH takes into account the most recent query access patterns over the database, and uses this information to auto-tune such that records that are accessed across similar sets of queries are hashed as much as possible to the same or nearby pages in the storage system. This property distinguishes Tunable-LSH from existing locality-sensitive hash functions, which are static. Our experiments with (i) an RDF data management system that uses Tunable-LSH, (ii) a hashtable that relies on Tunable-LSH to dynamically cluster its records, and (iii) workloads that rigorously test the sensitivity of Tunable-LSH verify the significant benefits of Tunable-LSH.

While the techniques presented in this paper facilitate the development of workload-adaptive RDF data management systems, some challenges are beyond the scope of this paper, hence, are left as future work.

First, this paper focuses on the question of "how" to tune but omits the question of "when". The question of "when" to tune the physical design of an RDF data management system is relevant because not all SPARQL workloads may exhibit the same degree of dynamism. Automatically detecting when changes occur in a workload can be an important step to eliminate or reduce redundant tuning steps (hence, the overhead of tuning).

Second, ideally, techniques are needed that can be used for tuning the RDF database after the execution of every query in the workload (i.e., to support extreme dynamism in workloads). While the techniques developed in this paper are more scalable than existing solutions, they support periodic runtime updates such as after the execution of every 10 or 100 queries. Extending these techniques to support more frequent runtime updates is also left as future work.

Third, the techniques proposed in this paper assume at least some predictability in workloads; more sophisticated predictive models (e.g., those that incorporate oscillations [40, 42]) can be developed in the future.

Lastly, it is worth exploring if Tunable-LSH can be used in a more general setting than just RDF systems. In fact, it should be possible to extend the idea of the self-clustering in-memory hashtable that we have implemented to a more general, distributed key-value store.

References

1. std::hash, 2015. `http://www.cplusplus.com/reference/functional/hash/`.
2. std::map, 2015. `http://www.cplusplus.com/reference/map/map/`.
3. std::unordered_map, 2015. `http://www.cplusplus.com/reference/unordered_map/unordered_map/`.
4. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18:385–406, 2009.
5. C. C. Aggarwal. A survey of stream clustering algorithms. In *Data Clustering: Algorithms and Applications*, pages 231–258. 2013.
6. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 496–505, 2000.
7. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 266–277, 1999.
8. R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.*, 25(3):355–380, 2016.
9. R. Al-Harbi, Y. Ebrahim, and P. Kalnis. Phd-store: An adaptive SPARQL engine with dynamic partitioning for distributed RDF repositories. *CoRR*, abs/1405.4979, 2014.
10. G. Aluç. *Workload Matters: A Robust Approach to Physical RDF Database Design*. PhD thesis, University of Waterloo, 2015. Available at: `https://uwspace.uwaterloo.ca/handle/10012/9774`.
11. G. Aluç, D. DeHaan, and I. T. Bowman. Parametric plan caching using density-based clustering. In *Proc. 28th Int. Conf. on Data Engineering*, pages 402–413, 2012.
12. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *Proc. 13th Int. Semantic Web Conference*, pages 197–212, 2014.
13. G. Aluç, M. T. Özsu, and K. Daudjee. Workload matters: Why RDF databases need a new design. *Proc. VLDB Endowment*, 7(10):837–840, 2014.
14. G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig. chameleon-db: a workload-aware robust RDF data management system. Technical Report CS-2013-10, University of Waterloo, 2013.
15. G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig. Executing queries over schemaless RDF databases. In *Proc. 31st Int. Conf. on Data Engineering*, pages 807–818, 2015.

16. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proc. 47th Annual Symp. on Foundations of Computer Science*, pages 459–468, 2006.
17. M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.
18. V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based hashing. In *Proc. 24th Int. Conf. on Data Engineering*, pages 327–336, 2008.
19. H. Bast and B. Buchhold. Qlever: A query engine for efficient sparql+text search. In *Proc. 27th ACM Int. Conf. on Information and Knowledge Management*, pages 647–656, 2017.
20. R. G. Bello, K. Dias, A. Downing, J. J. Feenan, Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *Proc. 24th Int. Conf. on Very Large Data Bases*, pages 659–664, 1998.
21. B. Berendt, L. Dragan, L. Hollink, M. Luczak-Rösch, E. Demidova, S. Dietze, J. Szymanski, and J. G. Breslin, editors. *Joint Proc. of the 5th International Workshop on Using the Web in the Age of Data and the 2nd International Workshop on Dataset PROFIling and fEderated Search for Linked Data*, volume 1362 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.
22. T. Bingmann. STX B+ tree C++ template classes, 2007. `https://panthema.net/2007/stx-btree/`.
23. B. Bislimovska, G. Aluç, M. T. Özsu, and P. Fraternali. Graph search of software models using multidimensional scaling. In *Proc. of the Workshops of the EDBT/ICDT 2015 Joint Conference*, pages 163–170, 2015.
24. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 121–132, 2013.
25. A. Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences*, pages 21–29, 1997.
26. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.
27. N. Bruno and S. Chaudhuri. To tune or not to tune? a lightweight physical design alerter. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 499–510, 2006.
28. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proc. 13th Int. World Wide Web Conf. - Alternate Track Papers & Posters*, pages 74–83, 2004.
29. S. Ceri, S. B. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.
30. M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th Annual ACM Symp. on Theory of Computing*, pages 380–388, 2002.
31. S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 3–14, 2007.
32. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. 20th Annual Symp. on Computational Geometry*, pages 253–262, 2004.
33. O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
34. H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proc. 17th Int. Conf. on Data Engineering*, pages 503–511, 2001.
35. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice (2^{nd} edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
36. K. R. French, G. W. Schwert, and R. F. Stambaugh. Expected stock returns and volatility. *Journal of Financial Economics*, pages 3–30, 1987.
37. L. Galarraga, K. Hose, and R. Schenkel. Partout: a distributed engine for efficient RDF processing. In *Proc. 23rd Int. World Wide Web Conf., Companion Volume*, pages 267–268, 2014.
38. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 518–529, 1999.
39. F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *Proc. VLDB Endowment*, 5(2):97–108, 2011.
40. G. Graefe, S. Idreos, H. A. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *Proc. Performance Evaluation, Measurement and Characterization of Complex Systems - 2nd TPC Technology Conf., TPCTC*, pages 169–184, 2010.
41. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300, 2014.
42. F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endowment*, 5(6):502–513, 2012.
43. R. W. Hamming, editor. *Coding and Information Theory*. Prentice-Hall, Englewood Cliffs, 1986.
44. R. Harbi, I. Abdelaziz, P. Kalnis, and N. Mamoulis. Evaluating SPARQL queries on massive RDF datasets. *Proc. VLDB Endowment*, 8(12):1848–1851, 2015.
45. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. W3C Recommendation, Mar. 2013.
46. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *Proc. 6th Int. Semantic Web Conference*, pages 211–224, 2007.
47. L. He, B. Shao, Y. Li, H. Xia, Y. Xiao, E. Chen, and L. Chen. Stylus: A strongly-typed store for serving massive RDF data. *Proc. VLDB Endowment*, 11(2):203–216, 2017.
48. K. Hose and R. Schenkel. WARP: workload-aware replication and partitioning for RDF. In *Proc. Workshops of the 29th IEEE Int. Conf. on Data Engineering*, pages 1–6, 2013.
49. M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *Proc. 21st Int. Conf. on Data Engineering*, pages 619–630, 2005.
50. M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. and Data Eng.*, 23(9):1312–1327, 2011.
51. S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *Proc. 3rd Biennial Conf. on Innovative Data Systems Research*, pages 68–78, 2007.

52. S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.
53. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annual ACM Symp. on Theory of Computing*, pages 604–613, 1998.
54. P. Jaccard. The distribution of flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912.
55. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. 31:264–323, 1999.
56. M. Kirchberg, R. K. L. Ko, and B.-S. Lee. From linked data to relevant data – time is the essence. *CoRR*, abs/1103.5046, 2011.
57. E. F. Krause, editor. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover, New York, 1986.
58. J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29:1–27, 1964.
59. V. I. Levenshtein. *Soviet Physics Doklady*, 10(8):707–710, 1966.
60. S. Lightstone, T. J. Teorey, and T. P. Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann, 2007.
61. J. P. McGlothlin and L. R. Khan. Materializing and persisting inferred and uncertain knowledge in RDF datasets. In *Proc. 24th Conf. on Artificial Intelligence*, 2010.
62. A. Morrison, G. Ross, and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2(1):68–77, 2003.
63. M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark - performance assessment with real queries on real data. In *Proc. 10th Int. Semantic Web Conference*, pages 454–469, 2011.
64. G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.
65. F. F.-H. Nah. A study on tolerable waiting time: how long are Web users willing to wait? *Behaviour & IT*, 23(3):153–163, 2004.
66. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
67. T. Neumann and G. Weikum. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endowment*, 3(1):256–263, 2010.
68. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *Proc. 21st Int. World Wide Web Conf., Companion Volume*, pages 397–400, 2012.
69. N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware, workload-adaptive SPARQL query caching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1777–1792, 2015.
70. W. Reed. The normal-Laplace distribution and its relatives. In *Proc. Advances in Distribution Theory, Order Statistics, and Inference*, pages 61–74, 2006.
71. T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proc. 19th Int. World Wide Web Conf.*, pages 851–860, 2010.
72. L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endowment*, 1(2):1553–1563, 2008.
73. Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3), 2010.
74. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endowment*, 1(1):1008–1019, 2008.
75. K. Wilkinson. Jena property table implementation. Technical Report HPL-2006-140, HP-Labs, 2006.
76. P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a fast and compact system for large scale RDF data. *Proc. VLDB Endowment*, 6(7):517–528, 2013.
77. L. Zeng and L. Zou. Redesign of the gStore system. *Frontiers Comput. Sci.*, 12(4):623–641, 2018.
78. D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 1087–1097, 2004.
79. L. Zou, J. Mo, D. Zhao, L. Chen, and M. T. Özsu. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endowment*, 4(1):482–493, 2011.