Algorithms and data structures for hyperedge queries

Jules Bertrand, Fanny Dufossé, Somesh Singh, Bora Uçar

▶ To cite this version:

Jules Bertrand, Fanny Dufossé, Somesh Singh, Bora Uçar. Algorithms and data structures for hyperedge queries. [Research Report] RR-9390, Inria Grenoble Rhône-Alpes. 2022, pp.28. hal-03127673v4

HAL Id: hal-03127673

https://hal.inria.fr/hal-03127673v4

Submitted on 28 Apr 2022

Algorithms and data structures for hyperedge queries

Jules Bertrand, Fanny Dufossé, Somesh Singh, Bora Uçar

RESEARCH REPORT
N° 9390
February 2021
Project-Teams ROMA, DataMove

ISSN 0249-6399   ISRN INRIA/RR--9390--FR+ENG

Algorithms and data structures for hyperedge queries

Jules Bertrand[*], Fanny Dufossé[†], Somesh Singh[‡], Bora Uçar[§]

Project-Teams ROMA, DataMove

Research Report n° 9390 — version 4 — initial version February 2021 — revised version April 2022 — 28 pages

Abstract: We consider the problem of querying the existence of hyperedges in hypergraphs. More formally, we are given a hypergraph, and we need to answer queries of the form "does the following set of vertices form a hyperedge in the given hypergraph?". Our aim is to set up data structures based on hashing to answer these queries as fast as possible. We propose an adaptation of a well-known perfect hashing approach for the problem at hand. We analyze the space and run time complexity of the proposed approach, and experimentally compare it with the state-of-the-art hashing-based solutions. Experiments demonstrate the efficiency of the proposed approach with respect to the state-of-the-art.

Key-words: Hashing, perfect hashing, hypergraphs

[*] ENS Lyon, 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.

[†] Inria Grenoble, Rhône Alpes, 38330, Montbonnot-Saint-Martin, France.

[‡] Inria Lyon and LIP (UMR5668 Université de Lyon - ENS Lyon - UCBL - CNRS - Inria), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France.

[§] CNRS and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes et structures de données pour le problème de requêtes d'existences d'hyperarêtes dans des hypergraphe

Résumé : Nous considérons le problème de requêtes d'existences d'hyperarêtes dans des hypergraphes. Plus formellement, pour un hypergraphe donné, nous devons répondre à des requêtes de la forme "est-ce que l'ensemble de sommets suivant forme une hyperarête de l'hypergraphe?". Notre objectif est de mettre en place une structure de donnée basée sur du hachage pour répondre à ces requêtes le plus rapidement possible. Nous proposons une adaptation d'une approche bien connue de hachage parfait pour notre problème. Nous analysons la complexité en temps et en espace de cette approche, et nous la comparons expérimentalement à des solutions de l'état de l'art basées sur le hachage. Les expériences démontrent l'efficacité de l'approche proposée par rapport à l'état de l'art.

Mots-clés : hachage, hachage parfait, hypergraphes

1 Introduction

Let $H = (V, E)$ be a hypergraph, where V is the set of vertices, and E is the set of hyperedges. We are given a set of queries of the form "is $h \subseteq V$ a member of E?", and we want to answer these queries. We are interested in data structures and algorithms enabling constant time per query in the worst-case with small memory requirements and construction/preprocess time. We focus on d-uniform, d-partite hypergraphs, where the vertex set is a union of d disjoint parts $V = \bigcup_{i=0}^{d-1} V^{(i)}$, and each hyperedge has exactly one vertex from each part $V^{(i)}$.

We are motivated by a tensor decomposition method proposed by Kolda and Hong [15]. This is a stochastic, iterative method targeting efficient decomposition of both dense and sparse tensors, or multidimensional arrays. Our focus is on the sparse case. For this case, Kolda and Hong propose a sampling approach, called stratified sampling, in which the nonzeros and the zeros of a sparse tensor are sampled separately at each iteration for accelerating the convergence of the decomposition method. The stratified sampling approach works as follows. The nonzeros of the input tensor are sampled uniformly at random. For sampling zeros in a d-dimensional tensor, a d-tuple of indices is generated randomly and tested to see if the input tensor contains a nonzero at that position. If the position is nonzero, the d-tuple is rejected and a new one is generated, until a desired number of indices corresponding to zeros of the input tensor are sampled. Sampling nonzeros is a straightforward task, as the nonzeros of a tensor are available, usually, in an array. To sample from the zeros of a tensor, Kolda and Hong propose a method based on sorting the nonzeros and then using binary search during query time to see if the tuple exists or not. They report that this approach is more efficient than other alternatives based on hashing in their tests—which are carried out in Matlab. Since the above implementation of sampling for zeros can be time consuming, Kolda and Hong propose and investigate other sampling approaches for their stochastic tensor decomposition method. Among the alternatives, the stratified sampling approach is demonstrated to be more useful numerically. That is why we are motivated to increase efficiency of the stratified sampling approach by developing data structures and algorithms for quickly detecting whether a given position in a tensor is zero or not.

A data structure that answers hyperedge queries can be used as a building block in a more general setting. For example, one can compute the number of edges or hyperedges contained in a given set of vertices in time polynomial in the size of the given vertex set, rather than in time proportional to the sum of the vertex degrees. For vertex sets of small cardinality with high vertex degrees, this becomes tangible.

Let $\mathcal{T}$ be a d-dimensional tensor of size $s_0 \times \cdots \times s_{d-1}$, where s_i is the size of the corresponding dimension. An entry in the tensor is indexed by a d-tuple, e.g., $\mathcal{T}[i_0, \ldots, i_{d-1}]$. One can associate a d-uniform, d-partite hypergraph $H = (V, E)$ with a tensor $\mathcal{T}$ as follows. In H, the vertex set is $V = \bigcup_{i=0}^{d-1} V^{(i)}$ where $V^{(i)} = \{v_0^{(i)}, \ldots, v_{s_i-1}^{(i)}\}$. Furthermore, there is a hyperedge $h \in E$ of the form $h = [v_{i_0}^{(0)}, \ldots, v_{i_{d-1}}^{(d-1)}]$ for each nonzero $\mathcal{T}[i_0, \ldots, i_{d-1}]$. From this correspondence, we see that the problem of testing if a given position in a tensor is zero can be cast as the problem of testing the existence of a hyperedge in the associated d-uniform, d-partite hypergraph.

We design and implement a perfect hashing based approach by building on the celebrated method by Fredman, Komlós, and Szemerédi [10] (FKS method). The FKS method stores a given set S of n elements with $O(n)$ space in such a way that it takes constant time to answer a membership query in the worst-case. The FKS method thus promises an asymptotically optimal solution to our problem of answering hyperedge queries. After reviewing the original FKS method in Section 2, we discuss the necessary changes to adapt it to answer hyperedge queries in Section 3. We first list some theoretical properties of the proposed method that are inherited from FKS in Section 3.1, for which the proofs are given in Appendix. We then present an approach to improve space utilization in Section 3.2; while our approach can also be used in the original FKS method, its effects are much more tangible in our use case. We note that since each element in our case is of size d, a lookup takes $O(d)$ time—which is not constant if d is part of the input. Since the queries are of size d, a query response time of $O(d)$ is optimal.

We restrict our attention to d-uniform, d-partite hypergraphs both in describing the proposed method and experimenting with it. This is so, as it covers the tensor decomposition application. Furthermore, as we discuss in Section 3.3.1, this is without loss of generality—the method is applicable to general hypergraphs.

To the best of our knowledge, the hyperedge query problem is first addressed by Kolda and Hong. The underlying problem is that of static hashing and hence existing perfect hashing methods can be used. Minimal perfect hash functions (MPHFs) are static data structures that map a given set with n elements to $\{0, \ldots, n-1\}$. By using MPHFs, one can store the ids of hyperedges in a space of size n and answer queries in constant time in the worst case. There are a number of publicly available MPHF implementations [9, 12, 18, 19, 22]. While these are highly efficient with practical implementations, the hyperedge query problem should be addressed on its own. This is so, as the d-dimensional structure of the hyperedges in our target application can enable special hashing methods, and converting the tensor's data into other structures for hashing affects the run time. One can also use approximate set membership filters based on Bloom filter and its variants [1, 8, 13, 16, 23]. Approximate set membership filters answer queries in such a way that the "no" answers are always correct while the "yes" answers could be incorrect (i.e., *false positives* are possible but *false negatives* are not). In order to use such filters for answering hyperedge queries or in other computations where exact answers are required, one has to double-check all the "yes" answer using an exact method. Therefore, a second exact hashing method needs to be built alongside the chosen approximate set membership method. Hence, such filters promise fast query response with an increased construction time and memory overhead.

We compare our approach experimentally (Section 4) with a number of methods that use the current state-of-the-art minimal and non-minimal perfect hashing methods, and a recent approximate membership filter. The most efficient version of the proposed method consistently outperforms all other methods considered. We note that there are recent studies which use random hypergraph models to build variants of Cuckoo hashing methods [2, 6, 12, 20, 28, 29]. Our work does not build random hypergraph models for designing hashing schemes; we seek static hashing methods for querying the existence of hyperedges in a given hypergraph.

2 Preliminaries and background

For an event $\mathcal{E}$, we use $\Pr(\mathcal{E})$ to denote the probability that $\mathcal{E}$ holds. For a random variable X, we use $\mathbf{E}(X)$ to denote the expectation of X. Markov's inequality states that for a random variable X that assumes only nonnegative values with expectation $\mathbf{E}(X)$, the probability that $X \geq c$ for a positive c is no larger than $\frac{\mathbf{E}(X)}{c}$, that is, $\Pr(X \geq c) \leq \frac{\mathbf{E}(X)}{c}$.

Let, $U = \{0, \ldots, u-1\}$ be the universe. Recall that a family of hash functions H from U to $\{0, \ldots, n-1\}$ is universal if for any $x \neq y \in U$, the probability that their key values are equal is bounded by $1/n$ [3]. In other words, $\Pr(h(x) = h(y)) \leq 1/n$ for a uniform random function $h \in H$. This definition can also be found in more recent treatments [24, Ch. 4].

We now give a brief summary of the hashing method by Fredman *et al.* [10] for static data sets. This method represents a given set of items using linear space and entertains constant time existence queries. We do not give the proofs, as some of our proofs for the proposed method in Section 3 follow closely that of Fredman *et al.* adapted to our case.

Let $U = \{0, \ldots, u-1\}$ be the universe and $S \subseteq U$ with $|S| = n$ be the set to be represented.

The FKS method [10] relies on a two-level approach for storing the set S. It needs a prime number p greater than $u-1$ and defines an extended universe $U' = \{0, \dots, p-1\}$. First, it chooses an element $k \in U'$ uniformly at random, and defines the *first level* hash function $h_k : U' \to \{0, \dots, n-1\}$ as $h_k(x) = (kx \bmod p) \bmod n$. It then assigns each element x of S to the bucket B_i where $i = h_k(x)$. As the outcome of the hashing function ranges from 0 to $n-1$, there are n buckets. Then, for a bucket B_i containing $b_i > 0$ elements, a storage space of size b_i^2 is allocated, a number $k^{(i)} \in U'$ is chosen at random, and the *second level* hash function $h_{k^{(i)}}(x) = (k^{(i)}x \bmod p) \bmod b_i^2$ is defined for items in B_i. A first requirement is that $\sum_i b_i^2$ should be $O(n)$ so that the method uses linear space. The second requirement is that each $k^{(i)}$ should be an injection for the respective bucket. In other words, two different elements in B_i should have different key values computed with the function $h_{k^{(i)}}(\cdot)$. If the hash functions are from a universal family, then both of these requirements can be met with high probability.

The FKS method constructs a representation of the given set by finding the values of k and $k^{(i)}$ respecting the constraints. These values are found with random sampling and trials. That is, the FKS method randomly chooses a $k \in U'$ and computes the size of the buckets when the first level hash function uses k. If the summation $\sum_i b_i^2$ is smaller than $3n$, then k is accepted, and the method proceeds to the second level. Otherwise, another k is randomly sampled and tried. A similar strategy is adopted for the hash functions in the second level. For the bucket B_i, a random $k^{(i)} \in U'$ is chosen, and the mapping defined by $(k^{(i)}x \bmod p) \bmod b_i^2$ is tested to see if it is an injection for the set of elements assigned to B_i. If so, that $k^{(i)}$ is accepted, if not, another one is sampled and tried. Fredman *et al.* note that with the bound $\sum_i b_i^2 < 3n$, and the injection requirement of b_i^2 space per bucket, one may end up testing many k and $k^{(i)}$. In order to have a construction time of $O(n)$ in expectation, they suggest testing with $\sum_i b_i^2 < 5n$ and using $2b_i^2$ space for each bucket B_i. Under these relaxations, at least one half of the potential k and $k^{(i)}$ values guarantee that the two requirements are met. The bounds $3n$ and $5n$ given in the original paper can be shown to be $2n$ and $4n$, when a universal hash function family is used.

In the FKS method, the membership query for an element $q \in U'$ can be answered in constant time by following the construction of the data structure. First, the bucket containing q is found by computing $i = (kx \bmod p) \bmod n$. If the bucket B_i is empty, then q is not in S. If B_i is not empty, the value $\ell = (k^{(i)}x \bmod p) \bmod 2b_i^2$ is computed, the element at location ℓ is compared with q, and the result of the comparison is returned as the answer to the query. The comparison between q and the element stored at the location ℓ is required as more than one element from U' can map to ℓ—whereas this can happen for only one element from S.

3 A lean variant of FKS

We discuss our adaptation of the FKS method for the hyperedge queries. We first discuss a family of hash functions for the two levels and show that the two requirements of having a total space of $O(n)$ and an injection for each bucket are met. We then propose two techniques for reducing the space requirement of the proposed method.

3.1 The hash function and its properties

In a d-partite d-uniform hypergraph $H = (V, E)$, the hyperedge set E is a set of d-tuples, which we will represent for hyperedge queries. Let n denote the number of hyperedges, and p be a prime number larger than n. Let U be the universe of all d-tuples of the form $[x_0, \dots, x_{d-1}]$ where x_i is between 0 and $p-1$; in other words $U = \{0, \dots, p-1\}^d$ and $E \subseteq U$. A potential approach to adapt the FKS for hyperedge queries is to convert d-tuples to unique integers by linearizing them. In this approach, in the case $d = 3$ for example, $[x_0, x_1, x_2]$ can be converted to $x_0 + s_0 \times (x_1 + s_1 \times x_2)$, where s_i corresponds to the size of dimension i; and a longer formula for higher d can be generated similarly. Afterwards, the FKS method can be used without any modification. Such an approach has limited applicability—the numbers get quickly too big for sparse tensors, as also noted by Kolda and Hong [15]. That is why we use d-tuples in defining the hash functions. Furthermore, since storing d-tuples in the buckets makes the storage requirement depend on d, we store the ids of the hyperedges in the buckets which are taken in the given order.

Let $\mathbf{x}, \mathbf{y}$ be two elements of the universe U. We use $\mathbf{x}^T\mathbf{y} = \sum_{i=0}^{d-1} x_i y_i$ to denote the inner product of the vectors corresponding to $\mathbf{x}$ and $\mathbf{y}$. In the proposed approach, as in the FKS method, a $\mathbf{k} \in U$ is chosen for the first level, and the hash function $h : U \to \{0, \ldots, n-1\}$ is defined as $h(\mathbf{x}) = (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. Then, each hyperedge $\mathbf{x} \in E$ is assigned to the bucket B_i where $i = h(\mathbf{x})$. We again use b_i to refer to the number of hyperedges from E that are mapped to B_i.

Lemma 1 below ensures that the linear space requirement can be met for any given set of hyperedges. We give the proof in Appendix A for completeness, where we also explain why the bound is $4n$, instead of $3n$ as in the original FKS method.

Lemma 1. *For a given set $E \subseteq U$ of n hyperedges, there is a $\mathbf{k} \in U$ such that when $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ is used as the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 4n$.*

Lemma 1 ensures the existence of a d-tuple resulting in a linear space, but does not specify how frequent such d-tuples are in the universe. The following corollary, whose proof is in the appendix, expresses a relaxation of the requirement on $\mathbf{k}$ as done by Fredman *et al.* to yield many candidates.

Corollary 2. *Let $E \subseteq U$ be a given set of n hyperedges. Then, for at least half of the potential $\mathbf{k} \in U$, when $\mathbf{k}$ is used in the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 7n$.*

Thanks to Corollary 2, one needs a constant number of trials, in expectation, to find a $\mathbf{k}$ respecting the space requirement of $\sum_{i=0}^{n-1} b_i^2 < 7n$.

We next show that for each bucket B_i, if we use a space of size b_i^2, we can map each element to a unique position with a function of the form $({\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p) \bmod b_i^2$. The proof is given in the appendix for completeness.

Lemma 3. *For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}' \in U$ such that the function $({\mathbf{k}'}^T\mathbf{x} \bmod p) \bmod b_i^2$ is an injection for $p \gg b_i^2$.*

While the bound $\sum_i b_i^2$ from Lemma 1 does not guarantee $p \gg b_i^2$, this must hold in practice for the original FKS and the proposed method to be efficient. A proposition below (Proposition 5) shows that there are many nonempty buckets, suggesting that a large b_i is unlikely for a random $\mathbf{k}$.

As done by Fredman *et al.*, one can relax the storage requirement of each bucket to have a constant number of trials in expectation to find a $\mathbf{k}'$ defining an injection. This is shown in the following corollary, whose proof is in the appendix.

Corollary 4. *Let B_i be a bucket with $b_i > 0$ elements. For at least half of the d-tuples $\mathbf{k}' \in U$, it holds that the function $({\mathbf{k}'}^T\mathbf{x} \bmod p) \bmod 2b_i^2$ defines an injection for the elements of B_i for $p \gg b_i^2$.*

3.2 Reducing the space requirements

While the previous lemmas show that we can use the FKS method with d-tuples $\mathbf{k}$ and $\mathbf{k}^{(i)}$ for each bucket B_i, there is a catch. For a bucket B_i, we need a space of size d to store $\mathbf{k}^{(i)}$. This

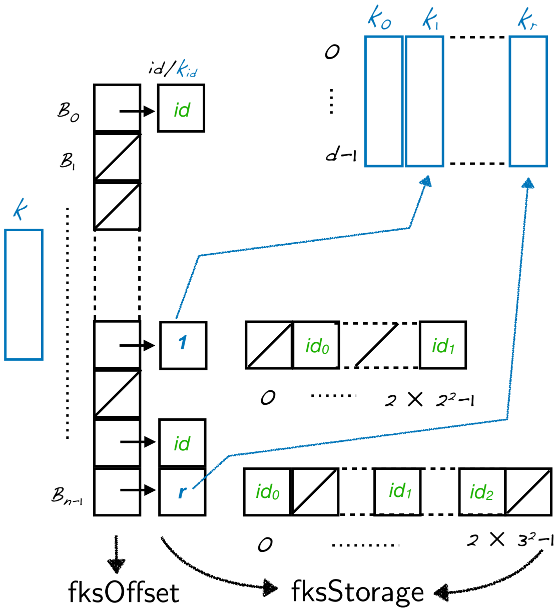


Figure 1 – In the proposed variant of FKS, the contents of a bucket B_i depend on the number b_i of hyperedges assigned to it. If b_i is 0, nothing is stored for B_i. If b_i is 1, the id of the hyperedge mapping to i is stored. If b_i is greater than 1, then the index of a suitable d-tuple from $\mathbf{k}_0$ to $\mathbf{k}_r$ in K is stored, along with a space of size $2b_i^2$ to hold the ids of b_i hyperedges assigned to B_i.

results in a space requirement of $O(nd)$ over all buckets. The space requirement will thus be large when the input tensor has a large number n of nonzeros or dimensions d.

An obvious way to reduce the space required to store the $\mathbf{k}^{(i)}$s is to avoid creating such a d-tuple for buckets with at most one element (those buckets B_i with $b_i = 0$ or $b_i = 1$). As we discuss in Proposition 5, one will still need, in expectation over all sets $F \subseteq U$ of n hyperedges, a total of $\Omega(nd)$ space.

Proposition 5. *For a random $\mathbf{k} \neq [0, \dots, 0]$ and a random set F with n hyperedges, the first level hash function using $\mathbf{k}$ creates at least $n\left(1 - e^{-1+2/p}\right)$ nonempty buckets in expectation, where e is the base of natural logarithm.*

The proof of this proposition can be found in the appendix.

In order to further reduce the memory requirements and obtain a lean variant of FKS for hyperedge queries, we propose to share the second level hash functions among the buckets. This can be achieved by storing a set K of d-tuples and keeping a reference to one suitable element of K for each bucket. The proposed variant of FKS is explained in Figure 1. We keep the first level the same as in the original FKS method and obtain a $\mathbf{k}$ in expected linear time that results in a suitable bound on the memory utilization as highlighted in Corollary 2. On the second level, rather than keeping a d-tuple per bucket we keep a set K of d-tuples with cardinality $|K|$ much smaller than n. The set K is such that for each bucket B_i with $b_i > 1$, there is at least one d-tuple in K defining an injection for the hyperedges in B_i, and B_i keeps a reference id to this tuple. The data structure fksStorage holds necessary space for all buckets, one after another. The data structure fksOffset holds the start of the storage space for each bucket. For an empty bucket B_i, nothing is stored and fksOffset[i] is nil; for a bucket B_i with one hyperedge, only the id of that hyperedge is stored in fksStorage and fksOffset[i] points to that position in fksStorage.

For a bucket B_i with $b_i > 1$ elements, fksOffset[i] points to the start of B_i in fksStorage. The space associated with B_i is of size $2b_i^2 + 1$ in which the id of a d-tuple in K is stored along with the ids of b_i hyperedges at suitable places.

We now discuss how to create the set K of tuples and bound the number of d-tuples we need to store in K. As the buckets independently need d-tuples for hashing, a random sequence of d-tuples generated for a bucket is also a random sequence of d-tuples for another one. With this observation in mind let us fix a random sequence of the d-tuples in the universe U. When we need a random d-tuple for a bucket, we try the d-tuples in the fixed random order, until we find one. We then include those d-tuples which were used by at least one bucket in K. Theorem 6 below shows that the number of d-tuples in K is $O(\log_2 n)$ in the expectation.

Theorem 6. *The size of K generated with the above technique will be smaller than $1 + \log_2 n$ in expectation for a given set E of n hyperedges.*

Proof. For each bucket, we know from Corollary 4 that at least half of the tuples from the universe U defines an injection. For a given bucket B_i, let X_i be a random variable counting the number of trials we did to find a suitable $\mathbf{k}^{(i)}$, where at the jth trial we consider the jth tuple from the fixed random sequence. Then,

$$\Pr(X_i = t) \le \frac{1}{2^t} \text{ for all } t \text{ and } i = 0, \ldots, n-1 \ .$$

Let $X_{i,t}$ be a random variable taking on value 1 if $X_i = t$, and 0 otherwise. Then, the expected number of buckets for which we did t trials to find an injection is

$$\mathbf{E}\left(\sum_{i=0}^{n-1} X_{i,t}\right) \le \frac{n}{2^t} \ ,$$

by the linearity of expectation. For a given t, the probability that there is a bucket for which we have found a second level hash at the trial t is

$$\Pr(X_i = t \text{ for some } i) = \Pr\left(\sum_{i=0}^{n-1} X_{i,t} \ge 1\right) \le \mathbf{E}\left(\sum_{i=0}^{n-1} X_{i,t}\right) \le \frac{n}{2^t} \ ,$$

by Markov's inequality.

Let us define another random variable

$$R_K = \max_i(X_i) \ ,$$

which corresponds to the maximum number of trials required until a second level hash function has been defined for all buckets, and therefore describes the number of d-tuples in K.

We will bound the expectation of R_K to obtain the bound stated in the theorem. We first note that

$$\begin{aligned} \Pr(R_K \ge r) = \Pr(X_i \ge r \text{ for some } i) \le \sum_{t=r}^{\infty} \Pr(X_i = t \text{ for some } i) &\le \sum_{t=r}^{\infty} \frac{n}{2^t} \\ &= \frac{n}{2^{r-1}} \ . \end{aligned} \tag{1}$$

The bound obtained in (1) is very large for small values of r. Indeed, when r is smaller than $\log_2 n$, 1 is a better bound on the probability. Therefore, we define a new random variable Y

which is equal to $\log_2 n$ if $R_K \leq \log_2 n$ and R_K otherwise. We will bound the expectation of Y. Since $\mathbf{E}(Y) \geq \mathbf{E}(R_K)$, the bound will also apply to the expectation of R_K and hence to the number of tuples in K. We have

$$\mathbf{E}(Y) = \log_2 n \Pr(R_K \leq \log_2 n) + \sum_{r=1+\log_2 n}^{\infty} \Pr(R_K \geq r) \leq \log_2 n + \sum_{r=1+\log_2 n}^{\infty} \frac{n}{2^{r-1}} ,$$

by using the bound (1) and the fact that $\Pr(R_K \leq \log_2 n) \leq 1$. Since

$$\sum_{r=1+\log_2 n}^{\infty} \frac{n}{2^{r-1}} = \frac{n}{2^{\log_2 n}} = 1$$

we obtain the result $\mathbf{E}(R_K) \leq \mathbf{E}(Y) \leq 1 + \log_2 n$. □

We note that Theorem 6 can also be useful to understand how far from its average value the number of elements in K can be. Indeed, we immediately deduce the following corollary from the proof above.

Corollary 7. *The probability that the number of d-tuples in K exceeds $t \log_2 n + 1$ is bounded by n^{1-t}.*

Another more intuitive way of seeing the $O(\log_2 n)$ bound of Theorem 6 is as follows. A randomly sampled $\mathbf{k}' \in U$ defines an injection with probability more than $1/2$ for a given bucket by Corollary 4. If we try a randomly sampled $\mathbf{k}'$ on all buckets, we will have an injection for half of the buckets in expectation. We can then randomly sample another d-tuple, which will again define an injection for half of the remaining buckets in expectation. By continuing this way, we see that $O(\log_2 n)$ tuples will thus be enough to define all injections, in expectation.

Based on Corollary 7, we suggest to create the set K to contain $2 \log n$ tuples at the outset. In the off chance that those tuples are not enough to define an injection for each bucket, new tuples can be created easily. The expected run time of construction can then be bounded as $O(nd)$. This is so since $\mathbf{k}$ can be found in expectation with two trials, where each trial is tested in $O(nd)$ time. Then $O(2d \log n)$ time is spent in creating K. By the intuitive explanation of Theorem 6, we see that another $O(nd)$ time is spent in perfectly hashing all buckets, leading to an overall $O(nd)$ run time bound in expectation.

A query for the existence of a hyperedge $\mathbf{q} \in U$ can be answered by first checking the size of the bucket B_i where $i = (\mathbf{k}^T\mathbf{q} \bmod p) \bmod n$. If $b_i = 0$, then $\mathbf{q}$ is not a hyperedge of the given hypergraph. If $b_i = 1$, the query is answered by comparing $\mathbf{q}$ with the hyperedge whose id is stored for B_i. If $b_i > 1$, then the associated d-tuple from K is retrieved as $\mathbf{k}^{(i)}$, and $\ell = (\mathbf{q}^T\mathbf{k}^{(i)} \bmod p) \bmod 2b_i^2$ is computed. If there is an id stored at the location ℓ, then the query is answered by comparing that hyperedge with $\mathbf{q}$. If there is no id at the location ℓ, then $\mathbf{q}$ is not in the hypergraph. A query can thus be read and answered in $O(d)$ time.

The space requirement can further be reduced by having more than n buckets; see the discussion following the proof of Lemma 1 given in Appendix and also the original FKS method [10, Section 4]. The gist of the idea is to reduce the number of items in the buckets so that fksStorage would need less total space. In the extreme case that we have n^2 buckets, fksStorage will just contain n entries with a suitable $\mathbf{k}$; however this time fksOffset will be of size n^2. On the other hand, with slightly more buckets than hyperedges, reductions in the space requirements will be observable in practice. Furthermore, with larger number of buckets than hyperedges, the number of buckets with at most one hyperedge will increase and hence the construction time and query response time are likely to reduce. In the experiments we analyze this parameter and find around 2.4 × n buckets to result in about the same space requirement when n buckets are used, while improving the construction and query response time.

3.3 Further discussions

3.3.1 Addressing general hypergraphs

We focus on the d-partite, d-uniform hypergraphs as this is a large class covering the requirements of the tensor decomposition application. The proposed method is not limited to this class. We can apply the algorithms to any hypergraph, without a requirement that the vertices belong to disjoint partitions or that the hyperedges are all of equal size. Let $H = (V, E)$ be a hypergraph and r be the maximum size of a hyperedge in E. Let us assume that 0 is not a member of V. We now define a new hypergraph $H' = (V', E')$ as follows. The vertex set $V' = \bigcup_{i=0}^{r-1} V^{(i)}$ where $V^{(i)} = V \cup \{0\}$ for $i = 0, \ldots, r-1$. The hyperedge set E' contains a hyperedge $\mathbf{e}'$ for each unique hyperedge $\mathbf{e} \in E$ of the original hypergraph. To create $\mathbf{e}'$ from $\mathbf{e}$, we first sort the vertices in $\mathbf{e}$ and use them in this order for the first $|\mathbf{e}|$ dimensions of $\mathbf{e}'$. For the dimensions $|\mathbf{e}|, \ldots, r-1$, we append a 0 to $\mathbf{e}'$ from the corresponding vertex partition $V^{(i)}$. The hypergraph H' is therefore r-uniform and r-partite. Any query hyperedge $\mathbf{q}$ for H can be converted similarly into a query hyperedge $\mathbf{q}'$ for H' by sorting its vertices, and adding the vertex 0 to $\mathbf{q}'$ for all missing dimensions $j = |\mathbf{q}|, \ldots, r-1$ so that $\mathbf{q}'$ becomes of size r as well. A query $\mathbf{q}$ posed on H can therefore be answered equivalently by the query $\mathbf{q}'$ on H'.

The above transformation of padding queries and hyperedges with 0 for missing positions should be done implicitly, otherwise the run time and space requirements can increase prohibitively. In particular, when processing a hyperedge in H', only the original vertices should be used while computing inner products with the $\mathbf{k}$ vectors. If the hyperedges and queries are not sorted at the outset, one needs to sort them, in which case query response time can increase in complexity.

3.3.2 Space complexity

The proposed method strives to achieve worst case optimal query response time on hypergraph data while maintaining the space requirement linear in the number n of hyperedges. Other perfect hash functions deal with the bits-per-key complexity, which measures the storage required to represent a minimum perfect hash function. The current state-of-the-art MPHFs generally target less than 3 bits/element. We do not concern ourselves with this complexity measure, as the motivating application stores the hyperedges with a total size of $O(nd)$ in order to answer queries for zeros of the tensor. In these cases, the bits-per-key complexity is not deemed to be an important aspect [18]. To store the hash functions of the proposed method, one at least needs to store $\mathbf{k}$, the number of elements, and the index of a d-tuple in K for each bucket with more than one element—one does not store the ids of hyperedges in buckets, see Figure 1. Since the number of d-tuples in K is $O(\log_2 n)$, one needs $\Omega(\log_2 \log_2(n))$ bits to store the id of a d-tuple per bucket. That is, the bits-per-key complexity will be $\Omega(\log_2 \log_2(n))$ to store the ids of d-tuples per bucket. This quantity is greater than 4 even when $n = 10^6$, which is already much larger than the bits-per-key complexity of the current state-of-the-art MPHFs. When this bit-complexity is too much, or when its implementation with standard data types requires too much space, then methods with smaller memory requirement are preferable.

4 Experiments

We compare the proposed algorithm called FKSlean with the following hashing methods, each of which uses a different hashing scheme:

- HashàlaFKS: the standard average-case constant time hashing method available in the

C++ standard library as unordered_map; we propose to use the first level hash function $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ of FKSlean with it. There is no second level hashing in HashàlaFKS. The tuple $\mathbf{k}$ used for HashàlaFKS enjoys the same properties as that of FKSlean in reducing the collisions (the number of hyperedges per bucket). We use HashàlaFKS as the baseline.

- BBHash [18]: a minimal perfect hash function. It has a parameter γ for which the original paper suggests values 1, 2 and 5, where $\gamma = 1$ optimizes space, $\gamma = 5$ optimizes the lookup time, and $\gamma = 2$ is in between. We use BBHash with $\gamma = 1$ and $\gamma = 5$.
- RecSplit [9]: another minimal perfect hash function. It has two parameters (LEAF_SIZE, bucket_size), where the configurations (8,100) and (5,5) are suggested in the original paper. We use RecSplit with these two configurations.
- PTHash [22]: the most recent minimal perfect hash function to the best of our knowledge, whose recent implementation [21] also creates non-minimal hash functions for efficiency. The original paper identifies four configurations which we use in our experiments: (1) C-C, $\alpha = 0.99$, $c = 7$, which optimizes the lookup time; (2) D-D, $\alpha = 0.88$, $c = 11$, which optimizes the construction time; (3) EF, $\alpha = 0.99$, $c = 6$, which optimizes the space effectiveness of the function; (4) D-D, $\alpha = 0.94$, $c = 7$, which optimizes the general trade-off. We use the version which creates non-minimal hash functions.
- FastFilter: This method uses one of the latest approximate set membership filters called *3-wise XOR binary fuse filters* [13] to filter out "no" answers and then needs to call one of the exact hashing methods for the "yes" answers.

In a preliminary set of experiments, we also tried a sort-based method, as it was used before in the original tensor decomposition application [15]. This method sorts the hyperedges in linear time using radix sort [4, Section 8.3], and then uses a binary search scheme to answer queries. We observed that the query time in this case was an order of magnitude larger than that of FKSlean on a large set of instances that we use in this section, and hence deemed it too slow. We thus do not give results with it. We note that comparison based search can be improved [14], these nonetheless will be inferior to constant time methods that we use in our experiments; especially since sorting and searching will need to work on d-dimensional data. In order to use the hashing methods BBHash, RecSplit, and PTHash in our context, we created the corresponding hash functions on the n hyperedges of a given hypergraph, and then used the resulting mapping function to uniquely store the id of each hyperedge in an array of suitable size. The resulting approaches are called uBBH, uRecSplit, and uPTHash, respectively, where the configurations are specified with (1) and (5) for BBHash, (8, 100) and (5,5) for RecSplit, and (1)–(4) for PTHash. All codes are compiled with g++ version 9.2 with options -O3, -std=c++17 -march=native as used in PTHash. We carry out the experiments on a machine having Xeon(R) CPU E7-8890 v4 with a clock-speed of 2.20GHz. All codes are available at https://gitlab.inria.fr/ucar/hedge-queries.

We first describe the data set, implementation, and the measurement details in the next two subsections. We then determine the *extension parameter* for FKSlean in Section 4.3. Using the determined parameter, in Section 4.4 we compare FKSlean with the other methods listed at the beginning of this section. We then investigate the space requirement of FKSlean in Section 4.5, and finally give a summary of experimental results in Section 4.6.

4.1 Data set

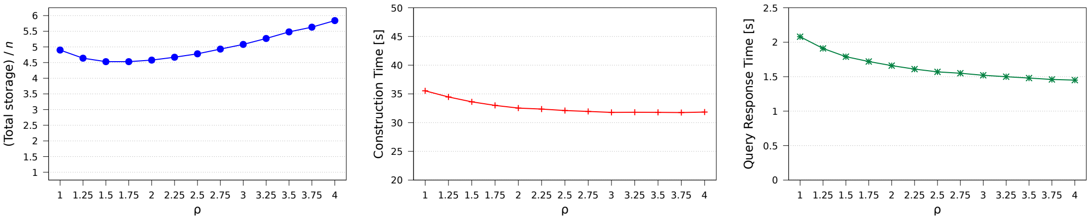
We present experiments both on real-life and synthetic data. We use the real-life data to compare the different algorithms, and use the synthetic data to investigate the behavior of different methods with respect to different problem parameters.

name	d	size in each dimension	n
kmer_A2a	2	170,728,175 ×170,728,175	360,585,172
queen_4147	2	4,147,110 × 4,147,110	329,499,288
com-Orkut	2	3,072,441 × 3,072,441	234,370,166
nell-1	3	2,902,330 × 2,143,368 × 25,495,389	143,599,552
delicious-3d	3	532,924 × 17,262,471 × 2,480,308	140,126,181
delicious-4d	4	532,924 × 17,262,471 × 2,480,308 × 1,443	140,126,181
flickr-3d	3	319,686 × 28,153,045 × 1,607,191	112,890,310
flickr-4d	4	319,686 × 28,153,045 × 1,607,191 × 731	112,890,310
nell-2	3	12,092 × 9,184 × 28,818	76,879,419
enron	4	6,066 × 5,699 × 244,268 × 1,176	54,202,099
vast-2015-mc1-3d	3	165,427 ×11,374 × 2	26,021,854
vast-2015-mc1-5d	5	165,427 × 11,374 × 2 × 100 × 89	26,021,945
chicago_crime	4	6,186 × 24 × 77 × 32	5,330,673
uber	4	183 ×24 × 1,140 × 1,717	3,309,490
lbnl-network	5	1,605 × 4,198 × 1,631 × 4,209 × 868,131	1,698,825

Table 1 – Real-life test data corresponding to the hypergraphs used in the experiments.

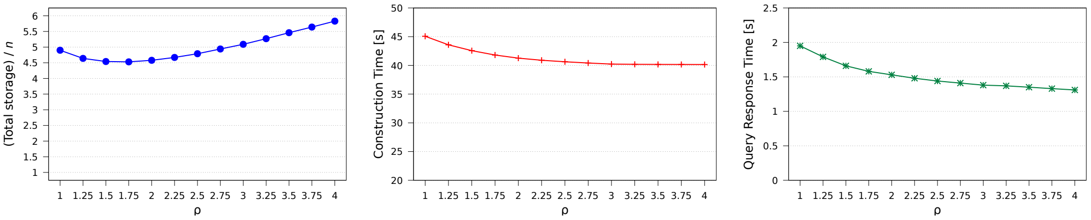
We take tensors from FROSTT [26], and build the associated d-uniform d-partite hypergraphs. The properties of the hypergraphs are shown in Table 1. The hypergraphs in the table are sorted in decreasing order of the number n of hyperedges. The tensors `delicious-3d` and `delicious-4d` contain the same data, with different dimensions. It turns out that the hyperedges are unique with or without the fourth dimension. Therefore, both hypergraphs have the same number of hyperedges. A similar observation is made concerning `flickr-*d`, while the number of hyperedges in `vast-2015-mc1-*d` differ only by 91. We also experiment with three bipartite graphs which correspond to real-life matrices available in The SuiteSparse Matrix Collection [5]. These are listed in the table with $d = 2$. These tensors and matrices arise in diverse real-world applications; ranging from natural language learning (`nell-*`), to e-mail data (sender-receiver-word-date in `enron`), and protein graphs (`kmer_A2a`) to social networks (`com-Orkut`).

The synthetic data are built using a model similar to the well-known Erdös-Renyi random graph model. Given a desired number d of parts, a desired number s of vertices in each part, and a desired number n of hyperedges, the model $\mathcal{R}(d, s, n)$ creates a random hypergraph as follows. First, n hyperedges are created by sampling their vertices in the ith part uniformly at random from the range $[0, s)$. Then, duplicate hyperedges are discarded. Note that the number of hyperedges can be slightly smaller than n, and that the maximum element in a part can be different from $(s - 1)$.



(a) Total storage requirement (b) Construction time (c) Query response time

Figure 2 – Analysis of the extension parameter ρ of FKSlean on nell-1.



(a) Total storage requirement (b) Construction time (c) Query response time

Figure 3 – Analysis of the extension parameter ρ of FKSlean on kmer_A2a.

4.2 Implementation and measurement details

In order to use BBHash, PTHash, and RecSplit we convert each input into 64-bit integers using SpookyHashV2. This function is also used by RecSplit implementation to handle keys different from 128-bit strings. We call SpookyHashV2 on each hyperedge by casting (no copy) the hyperedge to `char *`. We modified RecSplit to use such 64-bit integers as input (instead of bit-strings of length 128 bits). We use two arrays to implement FKSlean, as shown in Figure 1. Here, `fksStorage` is a contiguous array for storing all buckets one after another; `fksOffset` is an array of size $(n + 1)$, and `fksOffset[i]` stores the start of bucket B_i's storage in `fksStorage`.

We use fastmod library [17] for all $(\cdot \bmod n)$ operations, and fast modulo operations with Mersenne primes [27] to compute $(\cdot \bmod p)$ for $p = 2^{31} - 1$; one can also use the fastmod library for efficiency in case this p is not large enough. For the $\bmod 2b_i^2$ operations, we use the standard modulo operator `%` in C++. For reading the tensors from the disk, we use PIGO [11].

The construction time reported for FKSlean includes all the steps: finding the first level hash satisfying Lemma 1 or Corollary 2, allocating `fksOffset` and `fksStorage`, building the set K of $2\log(n)$ d-tuples, and setting up the data structure as shown in Figure 1.

We report the average of five runs in all measurements per hypergraph. A number q of queries are generated by mixing a set of existing hyperedges and a set of random hyperedges using a parameter t, called hit-ratio. First $\lfloor t \times q \rfloor$ hyperedges from the given hypergraph are chosen uniformly at random; then the remaining queries are generated by setting their ith coordinate with `rand()%s[i]` where `s[i]` is the size of the tensor in dimension i. At each of the five repetitions of the experiments, the same set of queries is used. The time to generate queries is not included in the query response time that we report.

4.3 Determining the extension parameter for FKSlean

As described in Section 3.2, the total storage requirement of FKSlean for the arrays fksOffset and fksStorage is contingent on the number of buckets. The higher the number of buckets, the less storage is needed for fksStorage. As an extreme, if we increase the number of buckets to n^2, then each bucket can have at most one element with a suitable **k**. Furthermore, the construction time and the query time are also affected: with more buckets, there will be more buckets with at most one element, and hence the first level hashing will suffice in more cases.

We study the impact of increasing the number of buckets on the storage requirement, and the performance of the construction and the query phases of FKSlean. Figures 2 and 3 show the effect of varying the number of buckets for two representative inputs from our test-suite. We set the number of buckets to ρn, where $\rho \geq 1$. We call ρ the *extension parameter*. As we can observe from Figures 2a and 3a, with increase in ρ, the total storage requirement of FKSlean initially decreases up to a point and then it increases. We also observe from Figures 2b and 3b that the construction time monotonically decreases with increase in ρ. Similarly, the query response time also goes down with increase in ρ as we observe in plots 2c and 3c. We perform this study on the synthetic data set of random hypergraphs as well, and observe similar trends; especially the total storage requirements were almost always identical to the plots in Figures 2a and 3a.

In an initial study, for $\rho = 1.0$, the storage requirement of FKSlean was observed to be slightly less than $5n$, which are also seen in Figures 2a and 3a. Based on the empirical evidence shown in Figures 2 and 3, we identify $\rho = 2.4$ as a sweet-spot for FKSlean. Fixing the number of buckets to $2.4n$ reduces the total storage requirement below $5n$ and reduces the construction and query time compared to that with n buckets. In the remainder of the experiments, we set the number of buckets to $2.4n$ for FKSlean unless otherwise stated.

4.4 Comparisons

4.4.1 With exact hashing methods

Table 2 presents the construction time of the ten methods on the real-life hypergraphs from Table 1. For every hypergraph, the absolute run time of HashàlaFKS is given in seconds, while the run time of the other methods are normalized with respect to HashàlaFKS. The last row of this table gives the geometric mean of the ratios of construction times to that of HashàlaFKS. As seen in this table, on all inputs, the construction time of all methods but FKSlean are longer than that of HashàlaFKS—as indicated by values greater than 1 for all the methods except FKSlean. FKSlean's construction time is 0.54 of HashàlaFKS's on average. FKSlean is the **fastest** method followed by uPTHash-(2).

Table 3 presents the query response times for the ten methods to answer 10^7 queries on hypergraphs from Table 1. In this table, the absolute query response time of HashàlaFKS is given in seconds. The response time of the other methods are normalized with respect to that of HashàlaFKS. Geometric mean of the ratios of the response time of different methods to that of HashàlaFKS are given in the last row. As seen in this table, among all methods, HashàlaFKS's query response is the largest, in general. FKSlean has the **fastest** query response on all the inputs, followed by uPTHash-(1) and uPTHash-(4). FKSlean is nearly 4 times faster than HashàlaFKS on average, and is nearly twice as fast, on average, compared to the next best performing methods.

In order to investigate how the methods behave with respect to the number of hyperedges and the dimension, we present further experiments with the random hypergraph family $\mathcal{R}(d, s, n)$. In the first set of experiments, we investigate how the run time of different methods change with the number n of hyperedges. To do so, we compare all methods on the random hypergraphs $\mathcal{R}(d, s, n)$ with $d = 4$, $s = 10^6$, and n taking different values in the range 10^6 to 5×10^8. Ideally,

		uRecSplit		uBBH		uPTHash				
name	HashàlaFKS (s)	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)	FKSlean
kmer_A2a	83.66	5.55	1.39	3.16	1.54	1.85	1.17	2.53	1.53	0.49
queen_4147	56.34	7.59	1.70	3.58	1.82	2.24	1.38	3.15	1.78	0.72
com-Orkut	51.74	5.71	1.25	2.49	1.22	1.54	1.00	2.23	1.28	0.56
nell-1	68.89	5.30	1.22	2.59	1.34	1.70	1.05	2.10	1.20	0.47
delicious-3d	64.71	5.49	1.22	2.49	1.30	1.59	1.02	2.22	1.32	0.55
delicious-4d	65.68	5.47	1.23	2.55	1.27	1.60	1.01	2.23	1.29	0.55
flickr-3d	51.59	5.52	1.19	2.45	1.18	1.56	0.99	2.22	1.27	0.55
flickr-4d	50.08	5.69	1.21	2.41	1.19	1.58	1.00	2.22	1.30	0.53
nell-2	34.05	5.66	1.15	2.41	1.00	1.48	0.97	2.16	1.25	0.57
enron	24.17	5.64	1.17	2.47	0.87	1.47	0.97	2.03	1.22	0.58
vast-2015-mc1-3d	10.68	6.12	1.17	2.56	0.82	1.44	0.96	1.92	1.20	0.64
vast-2015-mc1-5d	11.64	5.55	1.09	2.27	0.76	1.27	0.83	1.74	1.04	0.60
chicago-crime	1.63	8.13	1.44	3.20	0.97	1.47	1.10	1.90	1.29	0.36
uber	0.77	10.45	1.87	4.11	1.26	1.80	1.40	2.29	1.60	0.42
lbnl-network	0.25	16.45	2.85	6.25	1.98	2.53	2.08	3.12	2.29	0.63
geo-mean		6.57	1.36	2.88	1.19	1.65	1.10	2.24	1.37	0.54

Table 2 – The construction time for the ten methods on real-life tensors. The absolute run time of HashàlaFKS is given in seconds. The construction time of the other methods are normalized with respect to that of HashàlaFKS. Lower values imply better performance.

		uRecSplit		uBBH		uPTHash				
name	HashàlaFKS (s)	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)	FKSlean
kmer_A2a	6.93	0.76	0.73	0.84	0.81	0.44	0.52	0.54	0.47	0.21
queen_4147	6.35	0.73	0.72	0.80	0.77	0.46	0.53	0.56	0.48	0.24
com-Orkut	6.49	0.69	0.70	0.73	0.73	0.45	0.51	0.54	0.46	0.23
nell-1	6.85	0.69	0.69	0.75	0.75	0.50	0.51	0.54	0.46	0.24
delicious-3d	6.69	0.71	0.69	0.76	0.74	0.48	0.53	0.57	0.49	0.25
delicious-4d	6.81	0.74	0.77	0.81	0.78	0.63	0.69	0.65	0.64	0.28
flickr-3d	6.65	0.68	0.67	0.72	0.70	0.48	0.52	0.56	0.49	0.24
flickr-4d	6.49	0.72	0.74	0.77	0.76	0.63	0.69	0.65	0.64	0.28
nell-2	6.52	0.67	0.63	0.69	0.69	0.48	0.52	0.56	0.49	0.24
enron	6.60	0.69	0.68	0.70	0.69	0.59	0.63	0.62	0.60	0.28
vast-2015-mc1-3d	6.53	0.61	0.57	0.63	0.56	0.46	0.49	0.53	0.47	0.24
vast-2015-mc1-5d	6.71	0.63	0.61	0.65	0.60	0.52	0.56	0.58	0.53	0.30
chicago-crime	5.32	0.52	0.50	0.56	0.52	0.39	0.43	0.45	0.39	0.27
uber	4.75	0.48	0.44	0.51	0.52	0.32	0.36	0.39	0.33	0.25
lbnl-network	3.67	0.54	0.48	0.58	0.60	0.34	0.39	0.41	0.34	0.23
geo-mean		0.65	0.63	0.69	0.68	0.47	0.52	0.54	0.48	0.25

Table 3 – The query response time for 10^7 queries with the ten methods. For HashàlaFKS, its absolute query response time is given in seconds. The response time of the other methods are normalized with respect to that of HashàlaFKS. Lower values imply better performance.

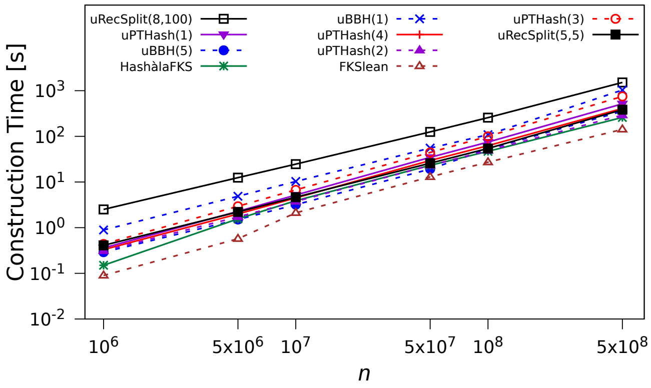


Figure 4 – The construction time of all methods on random hypergraphs in seconds. The x- and y-axes are in log-scale. At $n = 5 \times 10^8$, the plots correspond to, from top to bottom, the methods as listed in the legend (from left to right, top to bottom).

we expect the construction time to increase linearly with the increasing n, and the query response time to remain nearly constant, independent of n, for all the methods.

The plot for the construction times of the different methods on the random hypergraphs with $10^6 \leq n \leq 5 \times 10^8$ is shown in Figure 4. The x-axis the and y-axis are both in the log-scale. From the figure, we observe that FKSlean consistently has the shortest construction time compared to the other methods on random hypergraphs as before. We further observe that for most of the methods the construction time varies linearly with n, as expected in theory. For instance, for uBBH-(1) there is a ten-fold increase in the construction time (1s to 10s) as n goes from 10^6 to 10^7. For HashàlaFKS and FKSlean, the construction time does not always increase linearly with n, but is largely linear for the most part. For instance, for FKSlean, after $n = 10^7$ the construction time increases nearly linearly with n.

We next compare the query response times of different methods on the $\mathcal{R}(d, s, n)$ hypergraphs family with the same parameters as before. Figure 5 presents the results for the query response time; the x-axis is in the log-scale. From the figure, we reconfirm that FKSlean is consistently considerably faster than the others for all values of n. We also see that the query response time of all methods increases with the increasing number of hyperedges. This is because for larger n, the internal data structures take up more memory. As a result, the random accesses in the data structures increase the overall look up time.

We next investigate the behavior of the methods under study with respect to d. For this analysis, we pick HashàlaFKS, uRecSplit-(5,5), uPTHash-(2), uPTHash-(4), uBBH-(5), and FKSlean, since these methods have good performance so far. We analyse the performance of these methods in both the construction and the query phases. Figure 6a and Figure 6b show the construction and query response time of these methods on the random hypergraphs $\mathcal{R}(d, s, n)$ where $d = \{4, 8, 16\}$, $s = 10^6$, and $n = 2 \times 10^7$. We see in Figure 6a that the construction time of FKSlean is the least of all the remaining methods for all values of d, as before. We further observe that the construction time of uRecSplit-(5,5), uPTHash-(2), uPTHash-(4) and uBBH-(5) are very stable with the increasing value of d. This is because all these methods first map the input to a 64-bit integer before proceeding with the computation. As a result, the increase in the dimensions does not affect the performance of these methods after this mapping. On the other hand, we observe that for HashàlaFKS and FKSlean, the construction time increases with the

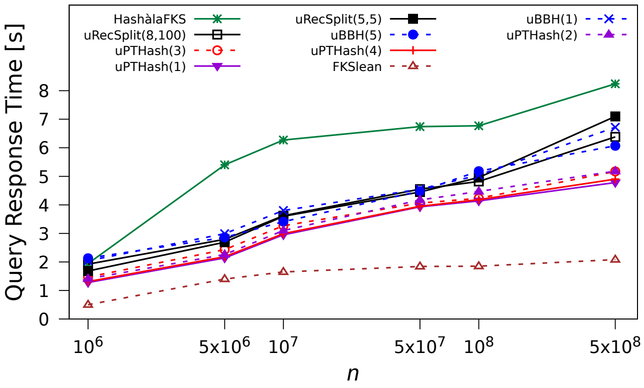


Figure 5 – The response time for 10^7 queries, in seconds, of all methods on the random hypergraphs. The x-axis is in log-scale. At $n = 5 \times 10^8$, the plots correspond to, from top to bottom, the methods as listed in the legend (from left to right, top to bottom).

increasing d. This is because for these methods, the work done per hyperedge increases with the increasing d. Figure 6b shows the query response time of the six methods on 10^7 queries. The query response time of FKSlean is the shortest for all values of d. Like their construction times, the query response times of uRecSplit-(5,5), uPTHash-(2), uPTHash-(4) and uBBH-(5), are stable with increasing d. Furthermore, the query response time of HashàlaFKS and FKSlean increase with the increasing d. This is again because of the increase in the work done per query with the increase in d.

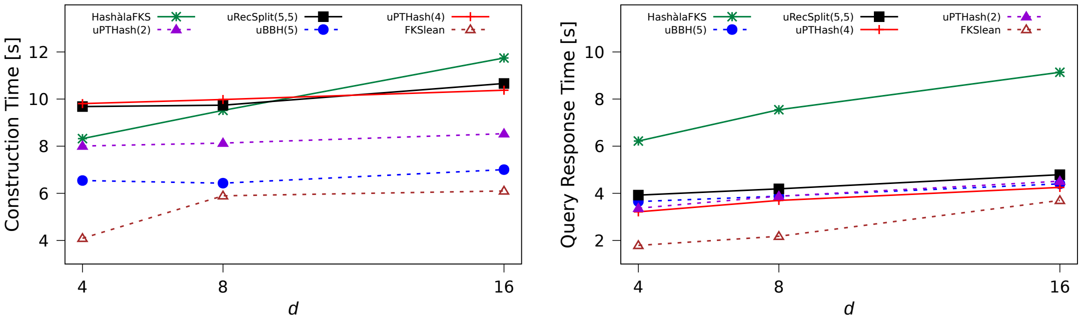
4.4.2 FKSlean vs FastFilter

In this section we compare FKSlean with FastFilter. FastFilter method uses *3-wise XOR binary fuse filters* first, and for all the "yes" answers it calls FKSlean (with $\rho = 2.4$), since this method is identified as the fastest in the previous subsection. In order to use FastFilter, we again use SpookyHashV2.

We compare FastFilter with two different variants of FKSlean—with $\rho = 1.0$, and $\rho = 2.4$. Figure 7 shows the comparison of query response times of FastFilter and FKSlean, for 10^7 queries, on two large inputs. We observe that for both nell-1 (7a) and kmer_A2a (7b), the query response time of FKSlean (with $\rho = 2.4$) is lower than FastFilter for all hit-ratios. On the other hand, the query response time of FKSlean (with $\rho = 1.0$) is higher than that of FastFilter till the hit-ratio reaches 0.6; thereafter, the former has a lower query response time.

We further zoom-in on the performance of FastFilter and observe that the lookup using *3-wise XOR binary fuse filters* takes up a significant chunk of the total query response time. For instance, for nell-1, the *3-wise XOR binary fuse filters* lookup takes 0.45 seconds out of the total query response time of 0.93 seconds, for hit-ratio $t = 0$. For kmer_A2a, the *3-wise XOR binary fuse filters* lookup takes 0.47 seconds out of the total query response time of 0.92 seconds, for hit-ratio $t = 0$.

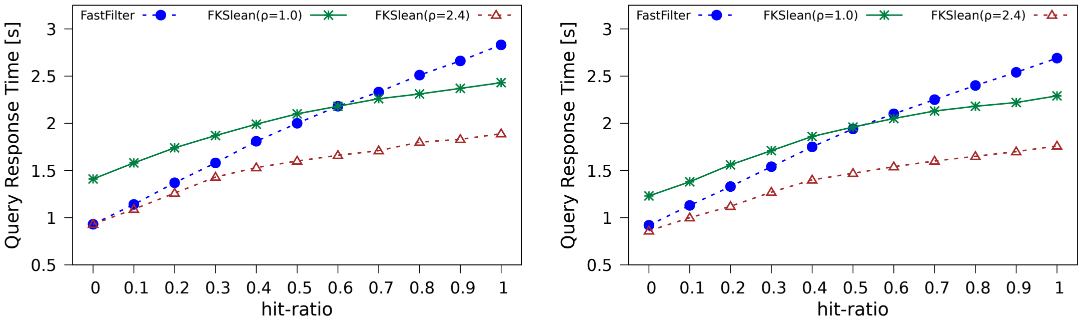
We next compare the construction times of FKSlean (with $\rho = 2.4$) and FastFilter. The construction time of FastFilter is always more than that of FKSlean. This is expected since FastFilter constructs *3-wise XOR binary fuse filters* and FKSlean data structure. We look closely at the time spent in the construction phase of the *3-wise XOR binary fuse filters* and FKSlean.



(a) Construction time

(b) Query response time

Figure 6 – The construction time and the query response time for 10^7 queries, in seconds, of six methods in hypergraphs from the family $\mathcal{R}(d, s, n)$ for $d = \{4, 8, 16\}$, $s = 10^6$, and $n = 2 \times 10^7$. At $d = 16$, the plots correspond to, from top to bottom, the methods as listed in the legends (from left to right, top to bottom).



(a) nell-1

(b) kmer_A2a

Figure 7 – Comparison of query response times of FKSlean and FastFilter for 10^7 queries on two instances.

H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15
4.73	4.70	4.74	4.74	4.74	4.74	4.73	4.73	4.74	4.73	4.73	4.73	4.75	4.51	4.65

Table 4 – The storage requirement of FKSlean for fksOffset and fksStorage arrays, normalized by n. The hypergraphs are labeled Hi, for $i = 1, \dots, 15$ and are given in the same order as Table 1.

We observe that for nell-1, the total construction time for the FastFilter is 52.80 seconds. Out of this, 21.73 seconds are spent in the construction of the *3-wise XOR binary fuse filters*, while 31.07 seconds are spent in constructing the FKSlean data structure. We further observe that for kmer_A2a, the total construction time of FastFilter is 65.63 seconds. Out of this, 25.16 seconds are spent in the construction of the *3-wise XOR binary fuse filters*, while 40.47 seconds are spent in constructing the FKSlean data structure. In general, across all the inputs, we find that constructing *3-wise XOR binary fuse filters* takes at least 60% of the construction time of the FKSlean, which attests the efficiency of FKSlean.

4.5 Storage Requirement of FKSlean

We now look at the storage requirement of FKSlean, and see how the theoretical properties shown in Lemma 1 and Proposition 5 compare with the results in practice. Table 4 shows the storage requirement of FKSlean for the arrays fksOffset and fksStorage for the input tensors from Table 1. The storage requirement, computed in terms of the number of cells of the two arrays, is normalized by the number n of hyperedges for each input data. From the table, we observe that the storage requirement of FKSlean is always comfortably less than $5n$ for all the inputs. The geometric mean storage requirement of FKSlean across all the input tensors is $4.71n$.

4.6 Summary of experimental observations

We summarize the findings of our experiments. Our extensive comparative study of the different methods shows that among all the methods, **FKSlean is always the best performing method**—it has the least construction time, as well as, the least query response time on all the real-life, and synthetic inputs. uPTHash is the next best method among the *exact* perfect hashing methods considered, both in construction and query response. However, we note that among the four variants of uPTHash, the same variant is not always the fastest in both construction and query. For instance, among the different variants of PTHash, uPTHash-(2) has the least construction time on average (cf. Table 2). On the other hand, uPTHash-(1) has the least query response time on average (cf. Table 3). Thus, the choice of the uPTHash variant needs to factor in the resulting trade-off between the construction time and the query response time.

The construction time of all exact methods varies nearly linearly with the number n of hyperedges (cf. Figure 4). The query response time of all methods increases gradually with increasing n (cf. Figure 5). The construction time and the query response time of uRecSplit, uPTHash and uBBH are unaffected by the dimension of the input hypergraphs. On the other hand, the performance of HashàlaFKS and FKSlean depends on the dimension of the input data (cf. Figure 6a and Figure 6b). Next, FKSlean (with $\rho = 2.4$) has a better query response than FastFilter for all hit-ratios (cf. Figure 7). Finally, HashàlaFKS's construction time is lower than all perfect hashing methods but FKSlean. However, its query response time is the largest among all the methods under study. Notwithstanding its inferior query response time, an advantage HashàlaFKS offers is simplicity and the ease of use. In order to use HashàlaFKS, one just needs to implement routines for determining a prime number $p > n$ and for computing $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. The `unordered_map` from C++ standard library takes care of the rest.

We note that the way the proposed FKSlean method handles the hyperedges natively makes a difference. To use BBHash, RecSplit, PTHash, and 3-wise XOR binary fuse filters one has to quickly convert the input data to 64-bit integers, or map the input data to a smaller universe (which is called universe reduction [27]). While this reduction/conversion does not take much time with the common libraries used, such as SpookyHashV2, it is not negligible. This is so even for FastFilter, as the proposed FKSlean (with the extension parameter $\rho = 2.4$) was demonstrated to be faster in query response. Apart from the overhead incurred during the universe reduction, there is also a theoretical issue associated with the worst case optimal hashing methods. Such reductions can create duplicate keys with a very small probability. In the presence of duplicate keys, the proposed FKSlean and other mentioned methods will have infinite loops unless the codes are modified to guard against the duplicates.

While FKSlean exhibits a superior performance than the other methods, it takes up more memory than the minimal and non-minimal perfect hashing methods considered in the experiments. On average FKSlean's storage requirement is $4.71n$ (cf. Table 4) unsigned integers (or the id type).

5 Conclusion

We investigated the problem of answering queries asking for the existence of a given hyperedge in a given hypergraph, with a special focus on d-partite, d-uniform hypergraphs arising in a tensor decomposition application. We proposed a perfect hashing method called FKSlean based on a well-known approach [10]. FKSlean has provably smaller space requirements than a direct adaptation of the original approach, thanks to the reuse of hash functions. Experimental results demonstrated in practice that the space requirement is in fact less than $5n$ plus an additional $O(d \log_2 n)$ term for storing the shared hash functions. We compared FKSlean with the methods using the current state-of-the-art minimal perfect hash functions (MPHFs), approximate set membership filters, and the `unordered_map` from C++ standard library equipped with the first level hash function used by FKSlean. Experiments on real-life and synthetic data showed that FKSlean achieves the shortest query response time among all alternatives while also having the least construction time of all. We have addressed the parallelization of the construction phase of FKSlean recently [25].

We have three lines of future work. On the application of interest with tensors, we need an implementation of the stochastic gradient method in an efficient library (instead of in Matlab) to test the effects of the proposed method in that particular application. On the more algorithmic front, we plan to address the dynamic case where hyperedges may get inserted or deleted. A suitable starting point is provided by Dietzfelbinger and others [7]. On the theoretical side, we observed that for any randomly chosen d-tuple $\mathbf{k}$ in the first level hashing, the quantity $\sum_i b_i^2$ was always comfortably smaller than the upper bound we have shown. Can a tighter upper bound be shown theoretically?

Acknowledgment

We thank Ioannis Panagiotas for his comments on a preliminary version of this report.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commununication of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782. doi: 10.1145/362686.362692.

[2] F. C. Botelho, R. Pagh, and N. Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[5] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[6] M. Dietzfelbinger and S. Walzer. Dense peelable random uniform hypergraphs. In M. A. Bender, O. Svensson, and G. Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[8] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer. Fast succinct retrieval and approximate membership using ribbon. *arxiv:2109.01892*, 2021.

[9] E. Esposito, T. Mueller Graf, and S. Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185, Philadelphia, PA, 2020. SIAM.

[10] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. ISSN 0004-5411.

[11] K. Gabert and U. V. Çatalyürek. PIGO: A parallel graph input/output library. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–279, Portland, OR, USA, 2021. IEEE, IEEE CPS. doi: 10.1109/IPDPSW52791.2021.00050.

[12] M. Genuzio, G. Ottaviano, and S. Vigna. Fast scalable construction of minimal perfect hash functions. In *15th International Symposium on Experimental Algorithms*, pages 339–352, St. Petersburg, Russia, 2016. Springer-Verlag.

[13] T. M. Graf and D. Lemire. Binary fuse filters: Fast and smaller than xor filters. *ACM J. Exp. Algorithmics*, 27, December 2022. ISSN 1084-6654. doi: 10.1145/3510449. URL `https://doi.org/10.1145/3510449`.

[14] P.-V. Khuong and P. Morin. Array layouts for comparison-based searching. *ACM J. Exp. Algorithmics*, 22, may 2017. ISSN 1084-6654. doi: 10.1145/3053370. URL `https://doi.org/10.1145/3053370`.

[15] T. G. Kolda and D. Hong. Stochastic gradients for large-scale tensor decomposition. *SIAM Journal on Mathematics of Data Science*, 2(4):1066–1095, 2020.

[16] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proc. VLDB Endow.*, 12(5):502–515, 2019. ISSN 2150-8097. doi: 10.14778/3303753.3303757. URL `https://doi.org/10.14778/3303753.3303757`.

[17] D. Lemire, O. Kaser, and N. Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *Softw. Pract. Exp.*, 49(6):953–970, 2019. doi: 10.1002/spe.2689.

[18] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-036-1. doi: 10.4230/LIPIcs.SEA.2017.25. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7619`.

[19] I. Müller, P. Sanders, R. Schulze, and W. Zhou. Retrieval and perfect hashing using fingerprinting. In J. Gudmundsson and J. Katajainen, editors, *International Symposium on Experimental Algorithms*, pages 138–149, Copenhagen, Denmark, 2014. Springer International Publishing.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Aarhus, Denmark, 2001. Springer Berlin Heidelberg.

[21] G. E. Pibiri and R. Trani. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *arxiv:2106.02350*, 2021.

[22] G. E. Pibiri and R. Trani. PTHash: Revisiting FCH minimal perfect hashing. In *44th SIGIR, International Conference on Research and Development in Information Retrieval*, pages 1339–1348, Virtual Event, Canada, 2021. ACM.

[23] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics*, 14, jan 2010. ISSN 1084-6654. doi: 10.1145/1498698.1594230. URL `https://doi.org/10.1145/1498698.1594230`.

[24] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox.* Springer International Publishing, 2019.

[25] S. Singh and B. Uçar. An Efficient Parallel Implementation of a Perfect Hashing Method for Hypergraphs. In *GrAPL 2022 - Workshop on Graphs, Architectures, Programming, and Learning*, pages 1–10, Lyon, France, May 2022. IEEE, IEEE CPS. URL `https://hal.inria.fr/hal-03612360`. to appear.

[26] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools. Available at `http://frostt.io/`, 2017.

[27] M. Thorup. High speed hashing for integers and strings. *arxiv:1504.06804*, 2015.

[28] S. Walzer. *Random hypergraphs for hashing-based data structures.* PhD thesis, Technische Universität Ilmenau, Germany, 2020.

[29] S. Walzer. Peeling close to the orientability threshold–spatial coupling in hashing-based data structures. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2194–2211, Virtual Conference, 2021. SIAM.

A Omitted proofs

We give the omitted proofs from Section 3. For convenience, we repeat the body of the lemmas and corollaries.

We start with Lemma 1 repeated below. Its proof follows closely the original proof by Fredman et al. and is given for completeness. After the proof, we explain why the upper bound is higher than that of the original theorem of Fredman et al.

▶ **Restatement of Lemma 1**: For a given set $E \subseteq U$ of n hyperedges, there is a $\mathbf{k} \in U$ such that when $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ is used as the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 4n$.

Proof. For a given $\mathbf{k}$, let $b_i^{(\mathbf{k})}$ denote the number of hyperedges in E having the same hash value $i = (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. The number of two-element subsets $\{\mathbf{x}, \mathbf{y}\}$ of E with the same hash value i is therefore $\binom{b_i^{(\mathbf{k})}}{2}$, that is $\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$. We will compute $\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ so that we can obtain the average number of two-element subsets of E with the same hash value over all potential $\mathbf{k}$ tuples. As there is at least one $\mathbf{k}'$ for which the corresponding sum $\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k}')}(b_i^{(\mathbf{k}')}-1)}{2}$ is no larger than the average, we will use that $\mathbf{k}'$ to attain the bound stated in the lemma.

We first observe that

$$\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} = \sum_{\substack{\mathbf{x},\mathbf{y}\in E\\ \mathbf{x}\neq\mathbf{y}}} \left|\{\mathbf{k}\in U : (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\}\right| , \qquad (2)$$

as the right hand side counts the total number of times any two different hyperedges $\mathbf{x}, \mathbf{y}$ of E have the same value for hash value over all $\mathbf{k} \in U$.

We will bound the right hand side of (2) from above. For this, we need a bound on the number of different $\mathbf{k}$ for which any $\mathbf{x}, \mathbf{y} \in E$ give the same value

$$(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\ . \qquad (3)$$

In other words, we want to count the number of different $\mathbf{k}$ for which we have

$$(\mathbf{k}^T\mathbf{x} - \mathbf{k}^T\mathbf{y}) \bmod p \in \left\{0, \pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\right\}\ . \qquad (4)$$

Since $\mathbf{x} \neq \mathbf{y}$, there is at least one dimension $0 \leq \ell < d$ such that $x_\ell \neq y_\ell$. Let us arbitrarily pick one such ℓ and write

$$\left(k_\ell(x_\ell - y_\ell) + \sum_{j\neq\ell} k_j(x_j - y_j)\right) \bmod p \in \left\{0, \pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\right\}\ . \qquad (5)$$

Note that the number of alternatives in the right hand side of (5) is less than $\frac{2p}{n} + 1$. Since we treat each pair $\mathbf{x}, \mathbf{y}$ once, we can assume $x_\ell > y_\ell$ without loss of generality. Let us take a value v from the right hand side of (5) and fix $\left(k_\ell(x_\ell - y_\ell) + \sum_{j\neq\ell} k_j(x_j - y_j)\right) \bmod p = v$. We will count the number of $\mathbf{k}$ tuples for which this equality holds. Then, multiplying with the upper bound on the number of elements in the right hand size, $\frac{2p}{n} + 1$, will give the total number of times any pair $\mathbf{x}, \mathbf{y} \in E$ satisfies (3).

At (5), we can freely set any k_j for $j \neq \ell$, and k_ℓ must then be chosen accordingly with these selections to make the equation hold. Because p is prime, for each distinct configuration of the k_j values for $j \neq \ell$, there is a unique value of k_ℓ that makes (5) hold. In other words, for any value in the right hand side of (4), there are p^{d-1} different $\mathbf{k}$ tuples, which are formed by considering all different p values for each k_j for $j \neq \ell$.

As there are no more than $\frac{2p}{n} + 1$ different right hand side values in (4), and for each one we have at most p^{d-1} different $\mathbf{k}$ tuples satisfying (3), we obtain an upper bound on the right hand side of (2) as

$$\sum_{\substack{\mathbf{x},\mathbf{y}\in S\\ \mathbf{x}\neq\mathbf{y}}} |\{\mathbf{k} \in U : (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\}| \leq p^{d-1}\left(\frac{2p}{n}+1\right)\frac{n(n-1)}{2} ,$$

since there are $\frac{n(n-1)}{2}$ pairs $\mathbf{x}, \mathbf{y}$.

Combining with the left hand side of (2), we see that

$$\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} \leq p^{d-1}\left(\frac{2p}{n}+1\right)\frac{n(n-1)}{2} .$$

Since there are p^d different $\mathbf{k}$, for at least one of them the inner sum $\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ should be no larger than the average. That is,

$$\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} \leq \frac{1}{p}\frac{2p+n}{n}\frac{n(n-1)}{2} , \tag{6}$$

for a $\mathbf{k}$. Let $\mathbf{k}'$ denote the tuple attaining that bound. Then,

$$\sum_{i=0}^{n-1} b_i^{(\mathbf{k}')}b_i^{(\mathbf{k}')} - \sum_{i=0}^{n-1} b_i^{(\mathbf{k}')} \leq 3(n-1) , \tag{7}$$

as $n < p$. Since $\sum_{i=0}^{n-1} b_i^{(\mathbf{k}')} = n$, we obtain the bound stated in the lemma. □

In the original FKS method, we have scalar values. Therefore the equivalent of (4) is

$$k(x-y) \bmod p \in \{0, \pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\} .$$

Since p is prime and both k and $x - y$ are less than p, the equality $k(x-y) \bmod p = 0$ cannot hold, and the set on the right hand side is $\{\pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\}$. This difference affects the upper bound, as obtained above.

Consider now that we have n' buckets with $n < n' < p$. In this case, the first level hashing will use $(\cdot \bmod p) \bmod n'$ to hash the hyperedges, and thus there will be $\frac{2p}{n'} + 1$ different right hand side values in (4). Following through the proof above we see that the upper bound in (7) reduces to $\frac{3n}{n'}(n-1)$. Thus, the total space requirement of individual buckets (fksStorage in Fig. 1) reduces, and the space requirements for holding the buckets (fksOffset in Fig. 1) increases.

► **Restatement of Corollary 2**: Let $E \subseteq U$ be a given set of n hyperedges. Then, for at least half of the potential $\mathbf{k} \in U$, when $\mathbf{k}$ is used in the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 7n$.

Proof. Let X be the random variable representing $\sum_i \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ when we randomly choose $\mathbf{k}$. Then, by (6) and the fact that $p > n$, we have $\mathbf{E}(X) \leq 3\frac{(n-1)}{2}$. By Markov's inequality, $\Pr(X \geq 3(n-1)) \leq \frac{\mathbf{E}(X)}{3(n-1)}$, and hence $\Pr(X \geq 3n) \leq \frac{1}{2}$. Therefore, we have $\Pr(X < 3n) \geq \frac{1}{2}$, and hence for at least half of the randomly chosen $\mathbf{k}$ we have $\sum_i \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} < 3n$. The event that $\sum_i b_i^2 < 7n$ is identical to the event $2X + n < 7n$, and hence holds with probability no smaller than $1/2$. Therefore, at least half of $\mathbf{k} \in U$ satisfies the bound of the corollary. □

▶ **Restatement of Lemma 3**: For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}' \in U$ such that the function $(\mathbf{k}'^T\mathbf{x} \bmod p) \bmod b_i^2$ is an injection for $p \gg b_i^2$.

Proof. The proof follows the same approach used in proving Lemma 1. There are two differences: here we have at most $2\left\lfloor \frac{p-1}{b_i^2} \right\rfloor + 1$ potential values for a pair to have an equal hash value (the equivalent of (4)), and the total number of pairs is $\frac{b_i(b_i-1)}{2}$ instead of $\frac{n(n-1)}{2}$. This leads to an average (over all possible $\mathbf{k}$) no larger than one for each position in the storage space of size b_i^2 for $p \gg b_i^2$. And hence, $\mathbf{k}'$ can be chosen. □

▶ **Restatement of Corollary 4**: Let B_i be a bucket with $b_i > 0$ elements. For at least half of the d-tuples $\mathbf{k}' \in U$, it holds that the function $(\mathbf{k}'^T\mathbf{x} \bmod p) \bmod 2b_i^2$ defines an injection for the elements of B_i for $p \gg b_i^2$.

Proof. We follow the proof of Lemma 1 (and Lemma 3), while replacing E by B_i, n by b_i, and by defining $b_j^{(\mathbf{k}')}$ to be the number of elements of B_i that map to $j \in \{0, \ldots, 2b_i^2 - 1\}$ with the function $(\mathbf{k}'^T\mathbf{x} \bmod p) \bmod 2b_i^2$. We obtain the inequality

$$\sum_{\mathbf{k}' \in U} \sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')} - 1)}{2} \le \frac{|U|}{p} \left(2\left\lfloor \frac{p-1}{2b_i^2} \right\rfloor + 1\right) \frac{b_i(b_i-1)}{2} \, .$$

By arithmetic simplification and using $p \gg b_i^2$, we obtain

$$\sum_{\mathbf{k}' \in U} \sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')} - 1)}{2} \le |U|\frac{1}{2} \, . \tag{8}$$

Let X be the random variable represeting $\sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')}-1)}{2}$ when we randomly choose $\mathbf{k}'$. By (8),

$$\mathbf{E}(X) \le \frac{1}{2} \, ,$$

over all potential $\mathbf{k}' \in U$. For a randomly chosen $\mathbf{k}'$ not to be an injection, $b_j^{(\mathbf{k}')} \ge 2$ must hold for some $j \in \{0, \ldots, 2b_i^2 - 1\}$. In that case, we will have the event $X \ge 1$. By Markov's inequality,

$$\Pr(X \ge 1) \le \mathbf{E}(X) \le \frac{1}{2} \, .$$

Therefore, $\Pr(X < 1) \ge \frac{1}{2}$, and hence at least half of the potential $\mathbf{k}' \in U$ defines an injection for B_i for $p \gg b_i^2$. □

▶ **Restatement of Proposition 5**: For a random $\mathbf{k} \neq [0, \ldots, 0]$ and a random set F with n hyperedges, the first level hash function using $\mathbf{k}$ creates at least $n\left(1 - e^{-1+2/p}\right)$ nonempty buckets in expectation, where e is the base of natural logarithm.

Proof. For each $\mathbf{x} \in U$ let us define a random variable $R_{\mathbf{x}}^{\mathbf{k}}$ taking on the value $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ for a tuple $\mathbf{k}$. We will compute for any tuple $\mathbf{k}$, the number of sets $F \subseteq U$ of cardinality n such that for all $\mathbf{x}$ in F we have $R_{\mathbf{x}}^{\mathbf{k}} \neq i$. We will obtain for each value i the probability over $\mathbf{k}$ and F that bucket B_i is non-empty. By summing over i, we will obtain the expected number of non-empty buckets for a randomly chosen $\mathbf{k}$ and a random set F of size n.

We first compute the expected value over $\mathbf{k}$ and F of the random variable $R_{F,i}^{\mathbf{k}} = |\{\mathbf{x} \in F | R_{\mathbf{x}}^{\mathbf{k}} \neq i\}|$. For a fixed tuple $\mathbf{k} \neq [0, \ldots, 0]$, let us find the number of $\mathbf{x} \in U$ such that $R_{\mathbf{x}}^{\mathbf{k}} = i$. If $R_{\mathbf{x}}^{\mathbf{k}} = i$, we should have

$$\mathbf{k}^T\mathbf{x} \bmod p \in \left\{ i, i \pm n, i \pm 2n, i \pm 3n, \ldots, i \pm \left\lfloor \frac{p-1-i}{n} \right\rfloor n \right\} . \tag{9}$$

This means that there are

$$t_i = 2 \left\lfloor \frac{p-1-i}{n} \right\rfloor + 1$$

possible values for $\mathbf{k}^T\mathbf{x} \bmod p$. For any value j in the right hand size of (9), let us consider the tuples $\mathbf{x}^j$ such that $\mathbf{k}^T\mathbf{x}^j \bmod p = j$. Since $\mathbf{k}$ is not uniformly zero, there is an index ℓ such that $k_\ell \neq 0$. Then, for any of the p^{d-1} possible values of x_r^j, where $r \neq \ell$, there exists one unique value of x_ℓ^j such that $\mathbf{k}^T\mathbf{x}^j \bmod p = j$. Thus, there exist p^{d-1} such $\mathbf{x}^j$ tuples for j. This yields a total of $p^{d-1}t_i$ alternatives for $\mathbf{x}$ such that (9) holds, and hence for which $R_{\mathbf{x}}^{\mathbf{k}} = i$, among all p^d elements of U.

There are therefore $\binom{p^{d-1}\cdot(p-t_i)}{n}$ sets F for which $R_{\mathbf{x}}^{\mathbf{k}} \neq i$, where the symbol $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ is the binomial coefficient. Then, for a fixed tuple $\mathbf{k}$, the probability that none of the elements of a random F maps to i is

$$\Pr(\cap_{\mathbf{x} \in F} R_{\mathbf{x}}^{\mathbf{k}} \neq i) = \frac{\binom{p^{d-1}\cdot(p-t_i)}{n}}{\binom{p^d}{n}} .$$

We can thus bound $\Pr(R_{F,i}^{\mathbf{k}} = 0)$ as follows:

$$\begin{aligned}
\Pr(R_{F,i}^{\mathbf{k}} = 0) &= \Pr(\cap_{\mathbf{x} \in F} R_{\mathbf{x}}^{\mathbf{k}} \neq i) \\
&= \frac{\binom{p^{d-1}\cdot(p-t_i)}{n}}{\binom{p^d}{n}} \\
&= \frac{\left(p^{d-1}\cdot(p-t_i)\right)!}{(p^{d-1}\cdot(p-t_i)-n)!} \frac{(p^d-n)!}{p^d!} \\
&= \prod_{j=0}^{n-1} \frac{\left(p^{d-1}\cdot(p-t_i)\right)-j}{p^d-j} \\
&= \prod_{j=0}^{n-1} \left(1 - \frac{p^{d-1}\cdot t_i}{p^d-j}\right) \\
&\leq \left(1 - \frac{p^{d-1}\cdot t_i}{p^d}\right)^n \\
&\leq \left(1 - \frac{t_i}{p}\right)^n \\
&\leq e^{-\frac{n\cdot t_i}{p}} \leq e^{-1+2/p},
\end{aligned}$$

as $\frac{n\cdot t_i}{p} \geq 2 - \frac{2}{p} - \frac{n}{p} \geq 1 - \frac{2}{p}$ by the definition of t_i and the fact that $p > n > i$.

By defining a binary variable $Y_i = 1$ if bucket B_i is empty and another one $Z = \sum_i Y_i$, we see that the expected number of empty buckets is

$$\mathbf{E}(Z) = \sum_{i=0}^{n-1} \mathbf{E}(Y_i) \leq ne^{-1+\frac{2}{p}} \ ,$$

for a randomly chosen $\mathbf{k}$ and a random set of n hyperedges, which concludes the proof. □

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399