ntHash: recursive nucleotide hashing

Hamid Mohamadi, Justin Chu, Benjamin P Vandervalk, Inanc Birol

Supplementary Data

ntHash computes the hash values for a given DNA/RNA sequence in a recursive way efficiently resulting in linear time hashing for sequences and huge improvement over the current hash methods used in genomics applications. To compute all hash values for a given DNA/RNA sequence we use:

$$H(k\text{-mer}_0) = rol^{k-1}h(r[0]) \oplus rol^{k-2}h(r[1]) \oplus \ldots \oplus h(r[k-1]) \quad (1)$$

$$H(k\text{-mer}_i) = rol^{1}H(k\text{-mer}_{i-1}) \oplus rol^{k}h(r[i-1]) \oplus h(r[i+k-1]),\ \mathrm{i}=1..l\text{-}k \quad (2)$$

where *rol* is a cyclic binary left rotation, ⊕ is the bit-wise exclusive or operator, and $h(\cdot)$ is a seed table where the letters of the DNA alphabet, Σ={A, C, G, T}, are assigned different random 64-bit integers. The related functions for the base and recursive parts of ntHash are:

```
inline uint64_t NT64(const char * kmerSeq, const unsigned k) {
    uint64_t hVal=0;
    for(unsigned i=0; i<k; i++)
      hVal ^= rol(h[(unsigned char)kmerSeq[i]], k-1-i);
      return hVal;
}

inline uint64_t NT64(const uint64_t hVal, const unsigned char charOut, const unsigned char
charIn, const unsigned k) {
    return (rol(hVal, 1) ^ rol(h[charOut], k) ^ h[charIn]);
}
```

To make hashing faster, in the implementation, we have replaced the *rol* operations with a pre-computed seed table of all possible shifts of the random 64-bit integers in *msTab*. Since there are 256 entries in the seed table and 64 possible shifts for each entry, the total size of the *msTab* is 256×64×64=1MB. The corresponding functions in the ntHash library are:

```
inline uint64_t NT64(const char * kmerSeq, const unsigned k) {
    uint64_t hVal=0;
    for(unsigned i=0; i<k; i++)
      hVal ^= msTab[kmerSeq[i]][(k-1-i)%64];
    return hVal;
}

inline uint64_t NT64(const uint64_t fhVal, const unsigned char charOut, const unsigned
char charIn, const unsigned k) {
    return(rol(fhVal, 1) ^ msTab[charOut][k%64] ^ msTab[charIn][0]);
}
```

To compute the canonical hash value of a given DNA/RNA sequence, we use:

$$H(k\text{-mer}'_0) = h(r[0]+d) \oplus rol^{1}h(r[1]+d) \oplus \ldots \oplus rol^{k-1}h(r[k-1]+d) \quad (3)$$

$$H(k\text{-mer}'_i) = ror^{1}H(k\text{-mer}'_{i-1}) \oplus ror^{1}h(r[i-1]+d) \oplus rol^{k-1}h(r[i+k-1]+d)\ ,\ \mathrm{i}=1..l\text{-}k \quad (4)$$

with the corresponding functions in the library:

```
inline uint64_t NTC64(const char * kmerSeq, const unsigned k, uint64_t& fhVal, uint64_t&
rhVal) {
    fhVal=0, rhVal=0;
    for(unsigned i=0; i<k; i++) {
      fhVal ^= msTab[kmerSeq[i]][(k-1-i)%64];
      rhVal ^= msTab[kmerSeq[i]+cpOff][i%64];
    }
    return (rhVal^fhVal);
}

inline uint64_t NTC64(uint64_t& fhVal, uint64_t& rhVal, const unsigned char charOut, const
unsigned char charIn, const unsigned k) {
    fhVal = rol(fhVal, 1) ^ msTab[charOut][k%64] ^ msTab[charIn][0];
    rhVal = ror(rhVal, 1) ^ msTab[charOut+cpOff][63] ^ msTab[charIn+cpOff][(k-1)%64];
    return (rhVal^fhVal);
}
```

Usually, we compute a single hash value when we hash a given *k*-mer. However, there are some bioinformatics applications utilizing the Bloom filter data structure that requires computing multiple hash values for a given *k*-mer. The existing hash methods do this by repeating the whole hashing procedure on a given *k*-mer with different initial seeds while in ntHahs we have provided a version to compute the multiple hash values in an efficient way. Given the number of required hash values, we first compute a hash value for the *k*-mer in the regular way using *NT64* function. We then perturb the 64-bit computed hash values by few additional operations without repeating the whole hashing procedure to get required number of hash values. The detailed procedure for the multi-hash version of ntHash is explained below. The uniformity results for the multi-hash version of ntHahs and other hash functions have been presented in Supp Figs. 2-5.

```
void NTM64(const char * kmerSeq, const unsigned k, const unsigned m, uint64_t *hVal) {
    uint64_t bVal=0, tVal=0;
    hVal[0] = bVal = NT64(kmerSeq, k);
    for(unsigned i=1; i<m; i++) {
        tVal = bVal * (i ^ k * multiSeed);
        tVal ^= tVal >> multiShift;
        hVal[i] =  tVal;
    }
}

void NTM64(const unsigned char charOut, const unsigned char charIn, const unsigned k,
uint64_t *hVal) {
    uint64_t bVal=0, tVal=0;
    hVal[0] = bVal = rol(hVal[0], 1) ^ msTab[charOut][k%64] ^ msTab[charIn][0];
    for(unsigned i=1; i<m; i++) {
        tVal = bVal * (i ^ k * multiSeed);
        tVal ^= tVal >> multiShift;
        hVal[i] =  tVal;
    }
}
```

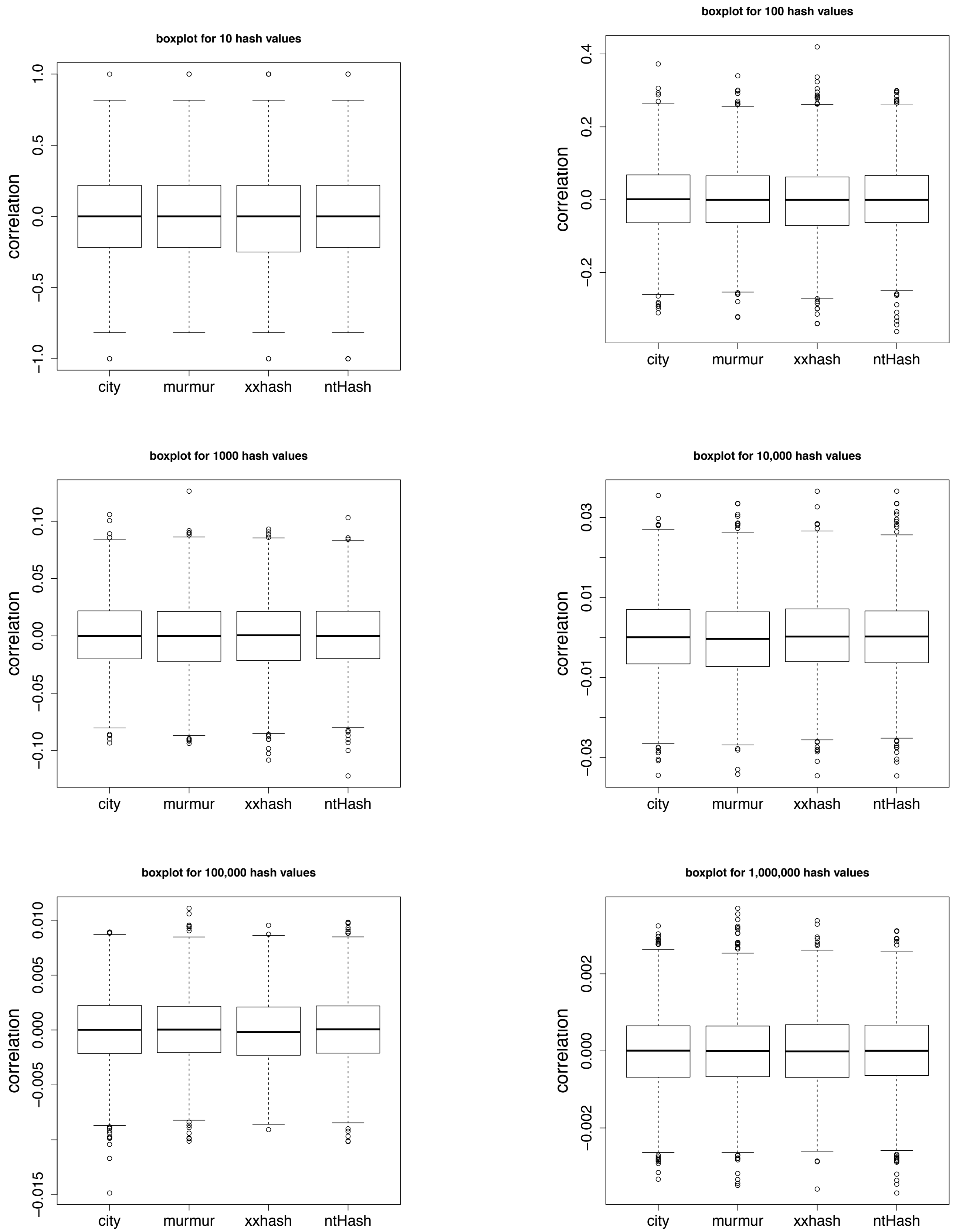
The methods we have used in the result section are:

- cityhash: https://github.com/google/cityhash
- murmur: https://github.com/aappleby/smhasher
- xxhash: https://github.com/Cyan4973/xxHash

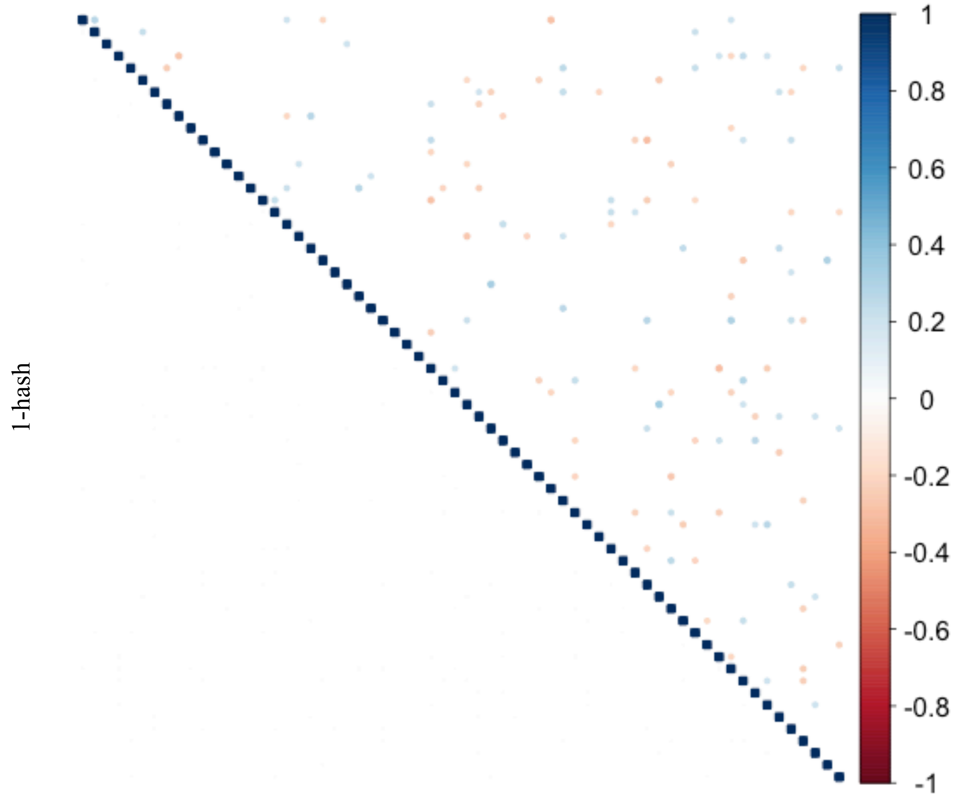
The data set we have used in the result section for Fig 1.a, b, c and Supp Tables 1-4 are from:

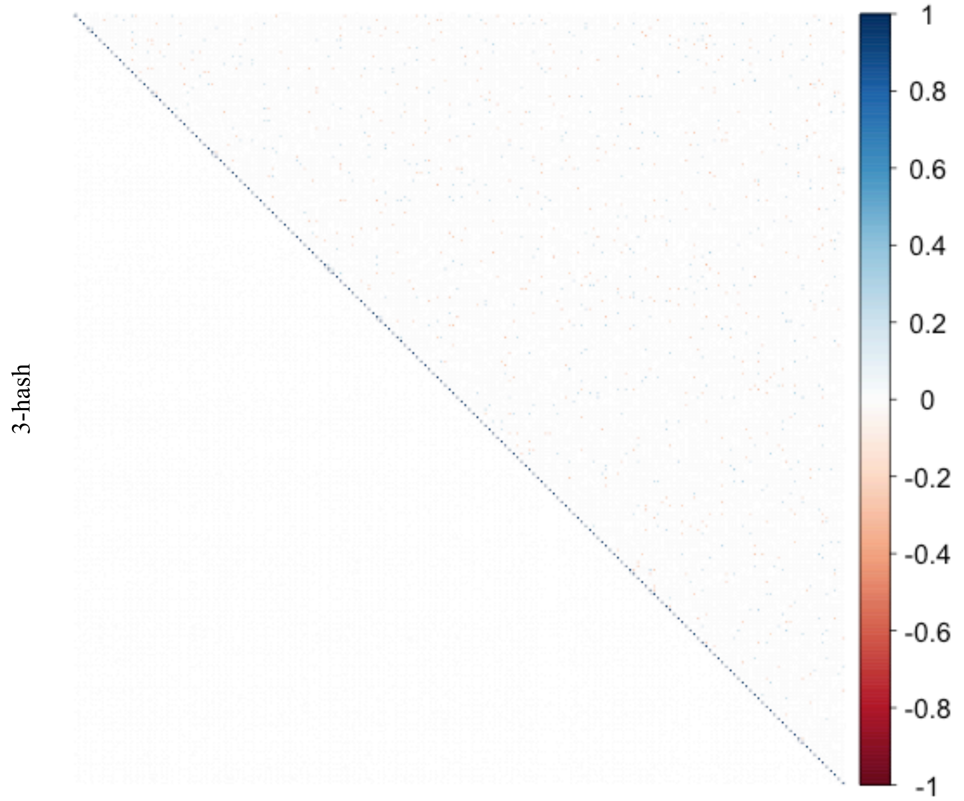
- Real data of Human individual NA19238 from Illumina Platinum Genome project http://sra.dnanexus.com/runs/ERR309932

For the others we have used randomly generated DNA sequences using seqgen.hpp in the ntHash package.

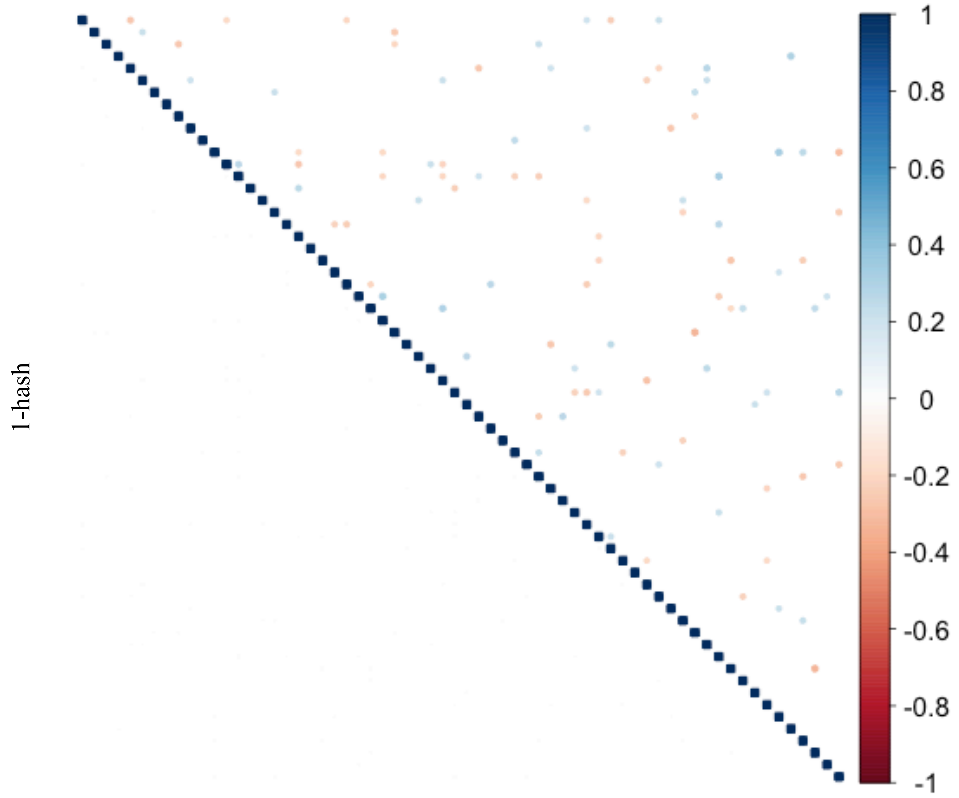


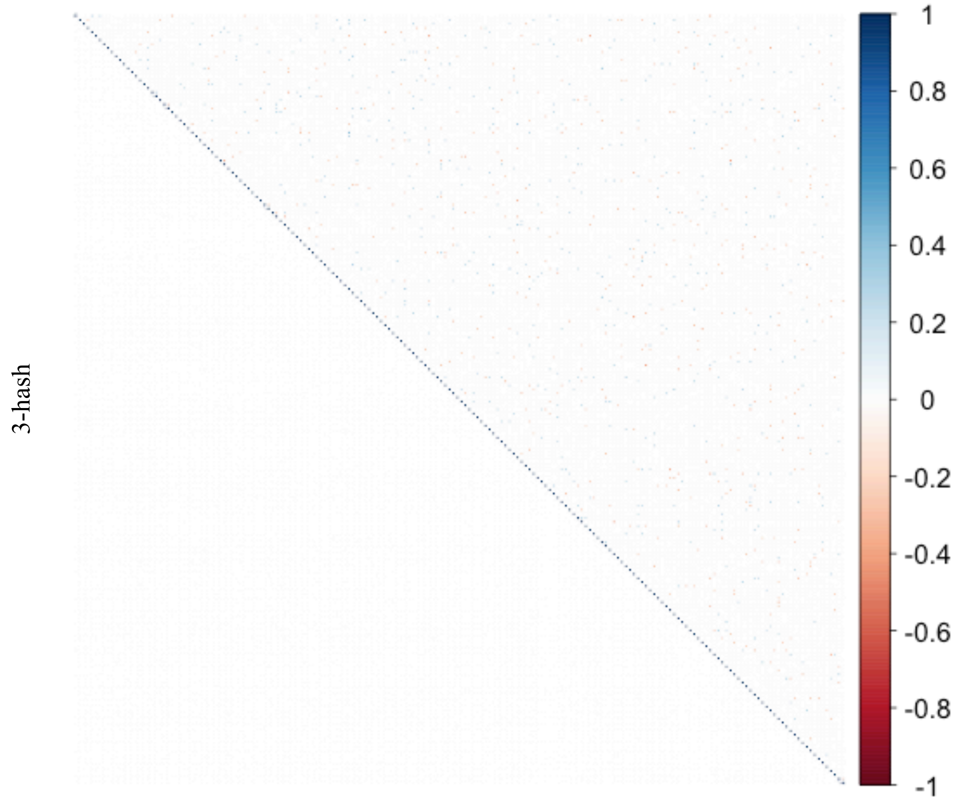
Supp. Fig 1. Correlation coefficient of hash algorithms for different sample sets. We see the correlation between hash value bits is near-ideal for ntHash as well as the state-of-the-art hashing methods like *cityhash*, *murmur*, and *xxhash*.



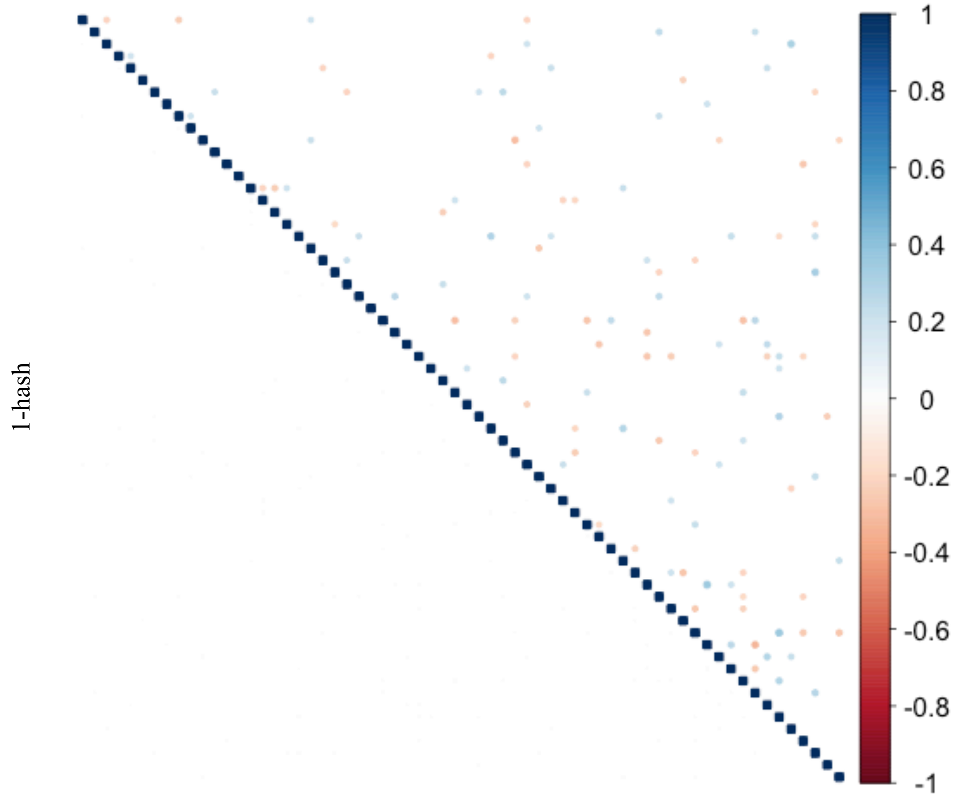


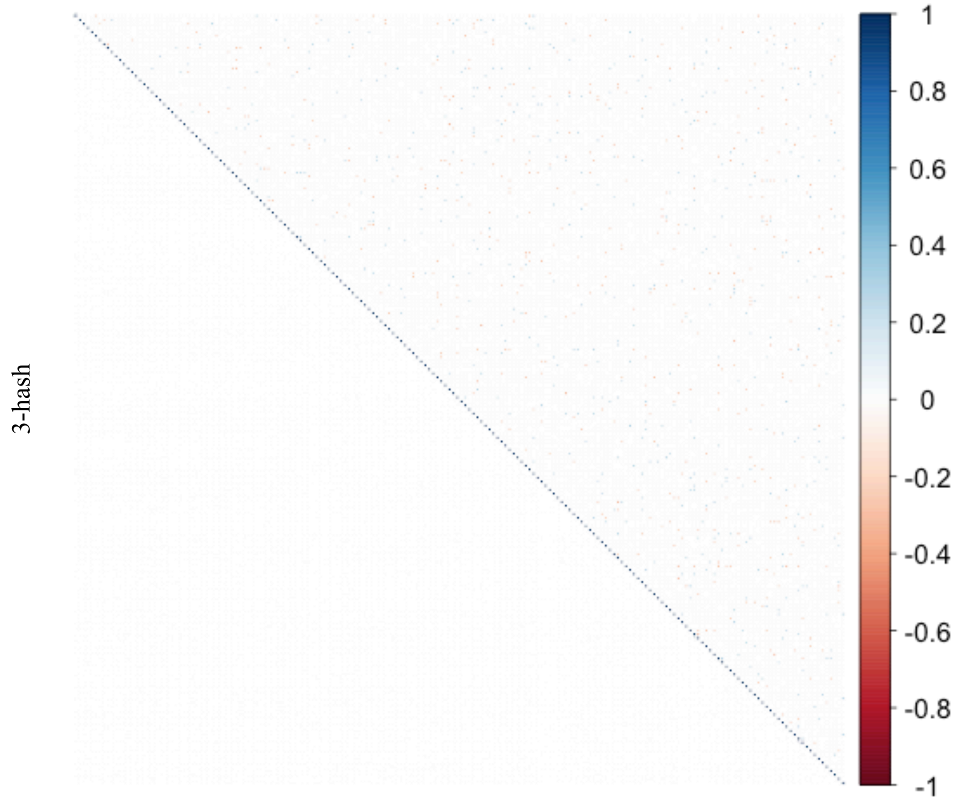
Supp Fig 2. Correlation coefficient plots of *cityhash* for one and three hashes on small (100 data points, the upper diagonal) and large sample set (100,000 data points, the lower diagonal).



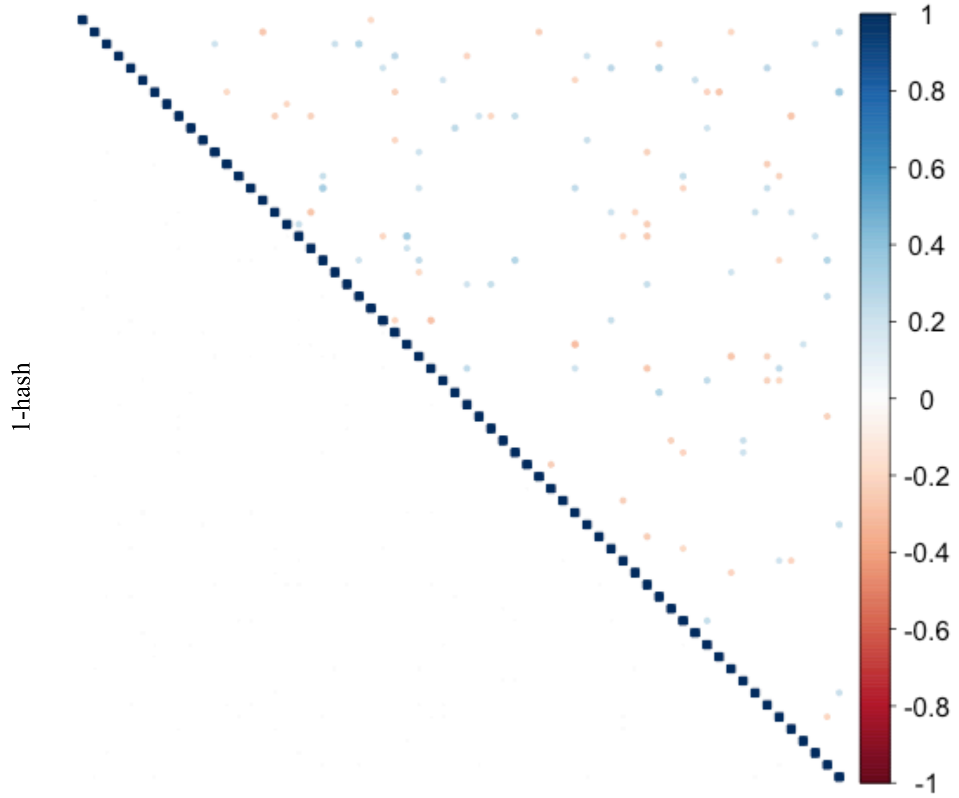


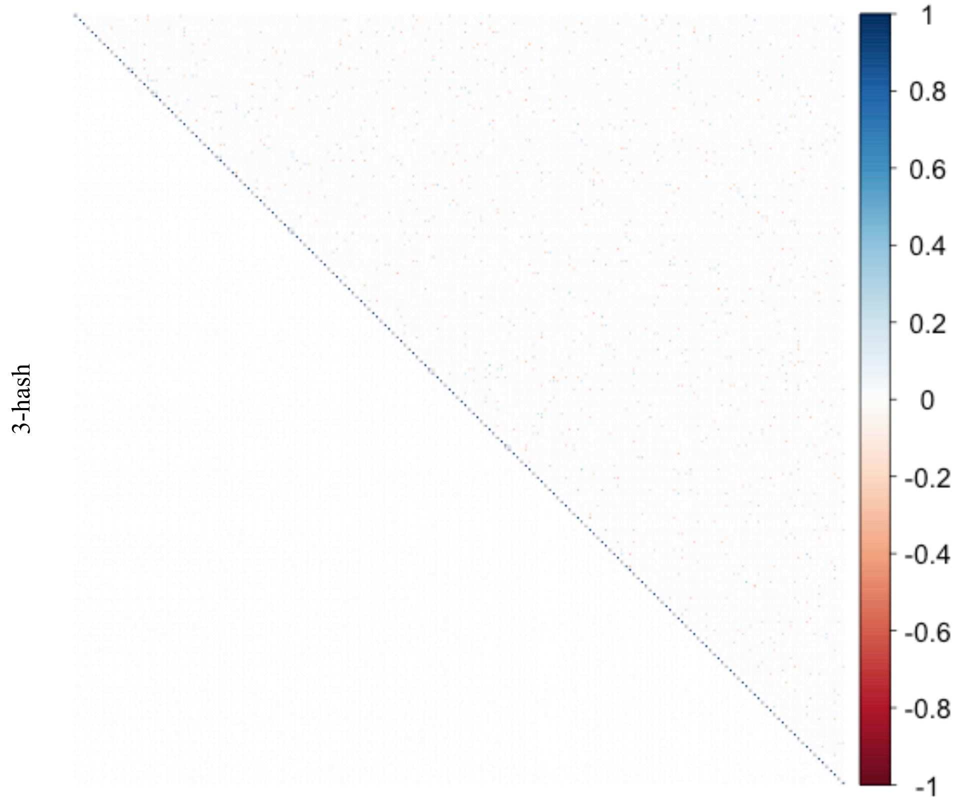
Supp Fig 3. Correlation coefficient plots of *murmur* for one and three hashes on small (100 data points, the upper diagonal) and large sample set (100,000 data points, the lower diagonal).



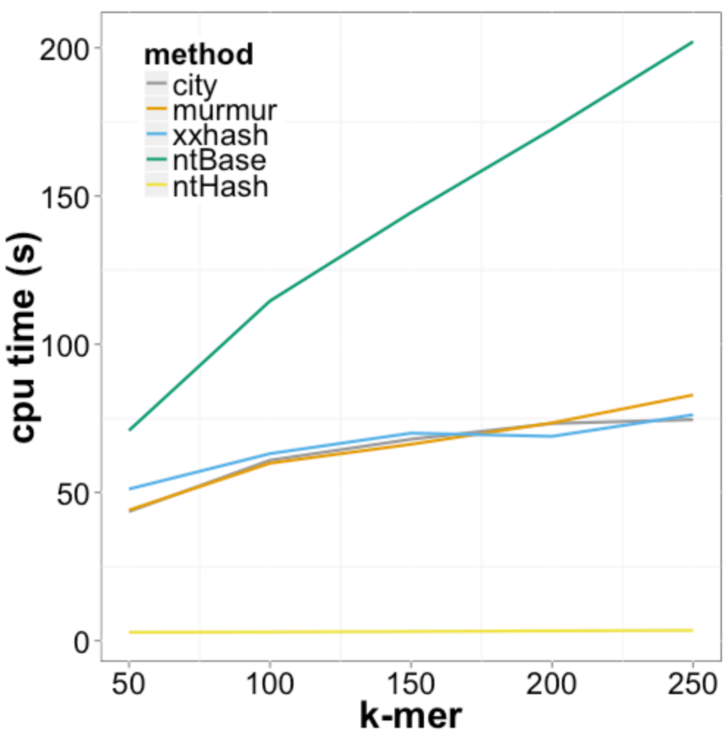


Supp Fig 4. Correlation coefficient plots of *xxhash* for one and three hashes on small (100 data points, the upper diagonal) and large sample set (100,000 data points, the lower diagonal).

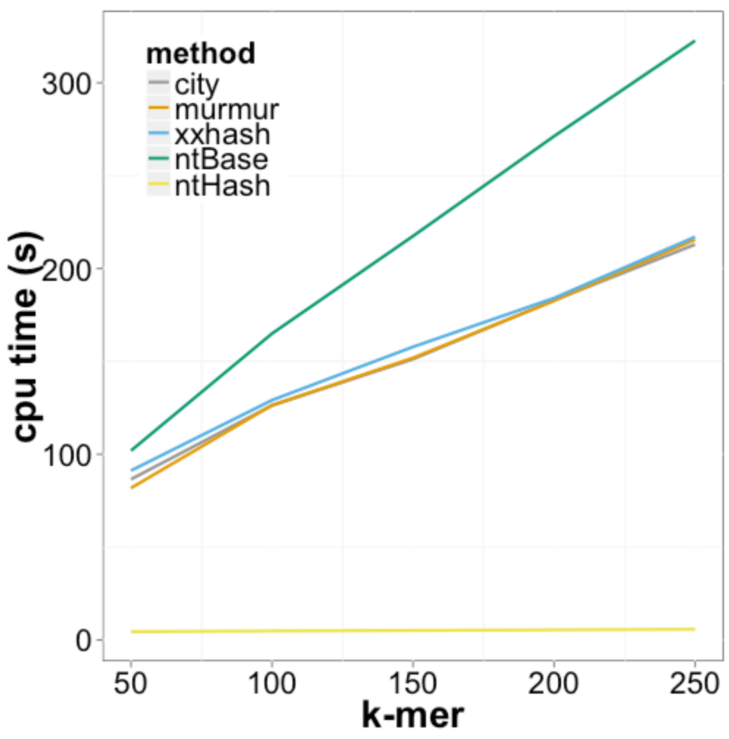




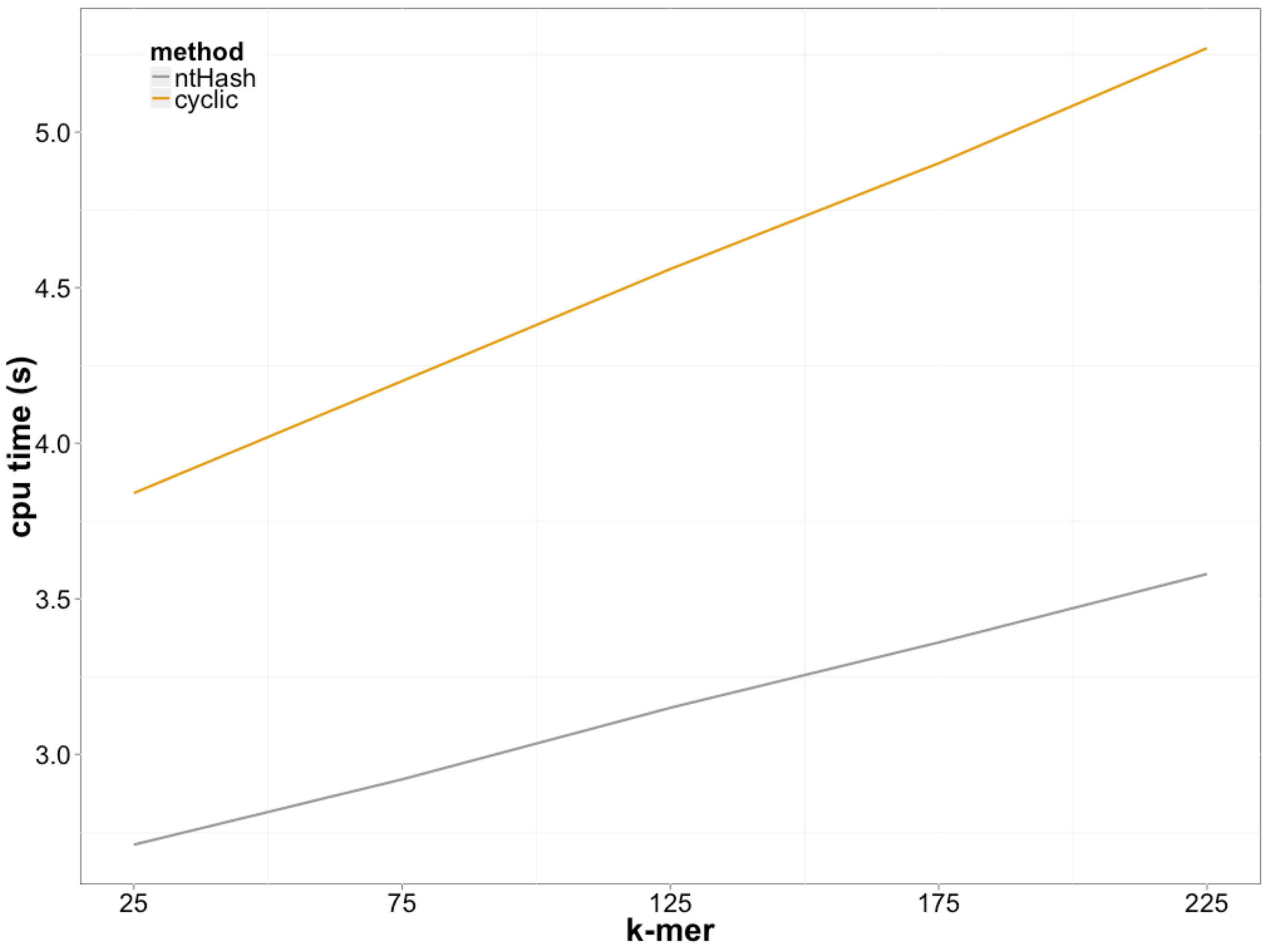
Supp Fig 5. Correlation coefficient plots of *ntHash* for one and three hashes on small (100 data points, the upper diagonal) and large sample set (100,000 data points, the lower diagonal).



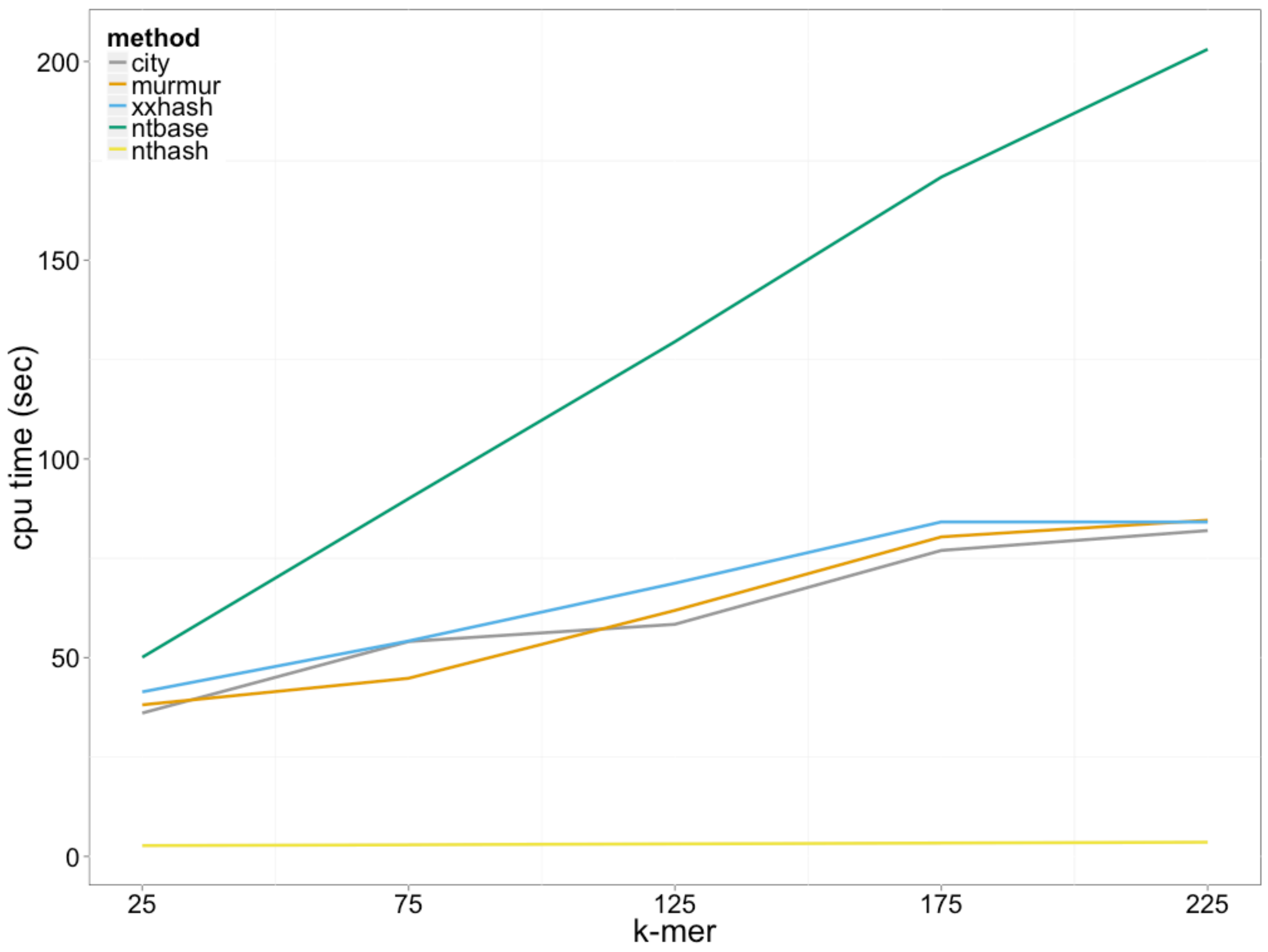
Supp Fig 6. CPU time for hashing 1 billion *k*-mers of lengths 50, 100, 150, 200, and 250. *ntBase* is the hash function based on non-recursive equation of ntHash, i.e. Equation (1).



Supp Fig 7. CPU time for canonical hashing 1 billion *k*-mers of lengths 50, 100, 150, 200, and 250. *ntBase* is the hash function based on non-recursive equation of ntHash, i.e. Equation (3).



Supp Fig 8. CPU time for hashing 1 billion *k*-mers of lengths 25, 75, 125, 175, and 225 using *ntHash* and *CyclicHash* from https://github.com/lemire/rollinghashcpp.



Supp Fig 9. CPU time for hashing 1 billion *k*-mers of lengths 25, 75, 125, 175, and 225. *ntBase* is the hash function based on non-recursive equation of ntHash, *i.e.* Supplementary Equation (1).

Supp Table 1. Bloom filter evaluation for cityhash on real data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470017244	804000000	94378728	11.74
	3	1250823778	804000000	24570368	3.06
	5	1858906097	804000000	17400323	2.16
k=150	1	469998731	404000000	47919015	11.86
	3	1250821595	404000000	12338460	3.05
	5	1858931600	404000000	8748217	2.17
k=250	1	469986185	4000000	472181	11.80
	3	1250802647	4000000	121776	3.04
	5	1858880965	4000000	85939	2.15

Supp Table 2. Bloom filter evaluation for murmur on real data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470007281	804000000	94295129	11.73
	3	1250852012	804000000	24560434	3.05
	5	1858955242	804000000	17381581	2.16
k=150	1	470002596	404000000	47411326	11.74
	3	1250839667	404000000	12338781	3.05
	5	1858923413	404000000	8750161	2.17
k=250	1	469997041	4000000	469325	11.73
	3	1250795983	4000000	122051	3.05
	5	1858914749	4000000	86649	2.17

Supp Table 3. Bloom filter evaluation for xxhash on real data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470016553	804000000	94292232	11.73
	3	1250840753	804000000	24518667	3.05
	5	1858942822	804000000	17394611	2.16
k=150	1	469995918	404000000	47411924	11.74
	3	1250830011	404000000	12340660	3.05
	5	1858930859	404000000	8743269	2.16
k=250	1	469986739	4000000	469547	11.74
	3	1250798629	4000000	122788	3.07
	5	1858874277	4000000	86494	2.16

Supp Table 4. Bloom filter evaluation for ntHash on real data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	469999521	804000000	94251857	11.72
	3	1250842928	804000000	24535428	3.05
	5	1858926469	804000000	17420006	2.17
k=150	1	469998307	404000000	47418185	11.74
	3	1250807514	404000000	12333989	3.05
	5	1858901461	404000000	8743479	2.16
k=250	1	469989680	4000000	469293	11.74
	3	1250786386	4000000	122542	3.06
	5	1858855900	4000000	86775	2.17

[*]. In all experiments, we first load the Bloom filter with 100 long sequences of length 5,000,000bp and allocating 8 bit/*k*-mer in Bloom filter. Next we query 4,000,000 real reads of length 250bp with *k*-mer of sizes 50, 150 and 250. The theoretical approximate false positive rate for *h*=1, 3, and 5 are 11.75%, 3.06% and 2.17%, respectively.

Supp Table 5. Bloom filter evaluation for cityhash on simulated data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470004731	804000000	94467037	11.75
	3	1250839836	804000000	24581360	3.06
	5	1858937046	804000000	17433637	2.17
k=150	1	470001865	404000000	47475319	11.75
	3	1250810005	404000000	12352225	3.06
	5	1858911301	404000000	8757714	2.17
k=250	1	469989444	4000000	469599	11.74
	3	1250773973	4000000	122939	3.07
	5	1858866870	4000000	87331	2.18

Supp Table 6. Bloom filter evaluation for murmur on simulated data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470017505	804000000	94484767	11.75
	3	1250827988	804000000	24589864	3.06
	5	1858925731	804000000	17424248	2.17
k=150	1	469997827	404000000	47469314	11.75
	3	1250784464	404000000	12347235	3.06
	5	1858877467	404000000	8757213	2.17
k=250	1	469992778	4000000	469332	11.73
	3	1250777246	4000000	121813	3.05
	5	1858869175	4000000	86187	2.15

Supp Table 7. Bloom filter evaluation for xxhash on simulated data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470008758	804000000	94470794	11.75
	3	1250821114	804000000	24585395	3.06
	5	1858965631	804000000	17434877	2.17
k=150	1	469991012	404000000	47464074	11.75
	3	1250815185	404000000	12350539	3.06
	5	1858920189	404000000	8751526	2.17
k=250	1	469997137	4000000	470919	11.77
	3	1250793131	4000000	122493	3.06
	5	1858893379	4000000	86618	2.17

Supp Table 8. Bloom filter evaluation for ntHash on simulated data[*].

k-mer	Hash	Set bit in BF	Query	False hit	FPR%
k=50	1	470001008	804000000	94473191	11.75
	3	1250822696	804000000	24594765	3.06
	5	1858911305	804000000	17428958	2.17
k=150	1	470009333	404000000	47464574	11.75
	3	1250812436	404000000	12351716	3.06
	5	1858915330	404000000	8757143	2.17
k=250	1	469979683	4000000	469658	11.74
	3	1250795888	4000000	122625	3.07
	5	1858897893	4000000	86664	2.17

[*]. In all experiments, we first load the Bloom filter with 100 long sequences of length 5,000,000bp and allocating 8 bit/*k*-mer in Bloom filter. Next we query 4,000,000 simulated reads of length 250bp with *k*-mer of sizes 50, 150 and 250. The theoretical approximate false positive rate for *h*=1, 3, and 5 are 11.75%, 3.06% and 2.17%, respectively.