Privacy-Preserving Search of Similar Patients in Genomic Data

Gilad Asharov*	Shai Halevi†	Yehuda Lindell‡	Tal Rabin†
Cornell Tech	IBM Research	Bar-Ilan University	IBM Research
asharov@cornell.edu	shaih@alum.mit.edu	Yehuda.Lindell@biu.ac.il	talr@us.ibm.com

February 15, 2017

Abstract

The growing availability of genomic data holds great promise for advancing medicine and research, but unlocking its full potential requires adequate methods for protecting the privacy of individuals whose genome data we use. One example of this tension is running Similar Patient Query on remote genomic data: In this setting a doctor that holds the genome of his/her patient may try to find other individuals with "close" genomic data, and use the data of these individuals to help diagnose and find effective treatment for that patient's conditions. This is clearly a desirable mode of operation, however, the privacy exposure implications are considerable, so we would like to carry out the above "closeness" computation in a privacy preserving manner.

Secure-computation techniques offer a way out of this dilemma, but the high cost of computing edit distance privately poses a great challenge. Wang et al. proposed recently [ACM CCS '15] an efficient solution, for situations where the genome sequences are so close that edit distance between two genomes can be well approximated just by looking at the indexes in which they differ from the reference genome. However, this solution does not extend well to cases with high divergence among individual genomes, and different techniques are needed there.

In this work we put forward a new approach for highly efficient secure computation for computing an approximation of the edit-distance, that works well even in settings with much higher divergence. We present contributions on two fronts. First, the design of an approximation method that would yield an efficient private computation. Second, further optimizations of the two-party protocol. Our tests indicate that the approximation method works well even in regions of the genome where the distance between individuals is 5% or more with many insertions and deletions (compared to 99.5% similarly with mostly substitutions, as considered by Wang et al.). As for speed, our protocol implementation takes just a few seconds to run on databases with thousands of records, each of length thousands of alleles, and it scales almost linearly with both the database size and the length of the sequences in it. As an example, in the datasets of the recent iDASH competition, it takes less than two seconds to find the nearest five records to a query, in a size-500 dataset of length-3500 sequences. This is 2-3 orders of magnitude faster than using state-of-the-art secure protocols for exact computation.

*Most of the work was done while the author was a post-doctoral researcher at IBM T.J. Watson Research Center. Previously supported by NSF Grant No. 1017660. Currently supported by a Junior Fellow award from the Simons Foundation.

†Supported in part by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office(ARO) under Contract No. W911NF-15-C-0236.

‡Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

Contents

1 Introduction

Consider the task of a medical doctor who wants to compare a patient's DNA against a remote genomic database, e.g., to determine the patient's pre-disposition to various medical conditions. The database contains a list of individual genome sequences, each labeled with the medical conditions of that person. The doctor needs to find the few individuals in the database whose genome sequence (in the relevant segment) most resembles that of the patient, and learn the medical conditions of these individuals. We define resemblance (or closeness) in terms of edit distance. Sending the patient's DNA sequence to the database has severe privacy implication, thus, we would like to find an effective privacy preserving solution to this task.

More specifically, we seek a solution to the following k-closest-match problem: We have a server holding a database DB of genomic sequences $(S_1, \ldots, S_m)$, whose approximate length and position inside the human genome are known. The client (doctor) is holding a sequence query Q, and wishes to find the identities of the k sequences in the DB that have the smallest edit distance from Q (where k is a public parameter). The goal is to perform this computation in a privacy-preserving manner. We target security in the presence of an honest-but-curious adversary. Our work was motivated by the recent "secure genome analysis competition" run by iDASH [15] (in which we won the 1st place for accuracy and speed).

Unfortunately, the straightforward solution of computing the exact edit distance [21] using a secure-computation protocol (or even the near-linear-time approximation of Andoni and Onak [3]) will be prohibitively slow. Even using state-of-the-art secure-computation techniques, such protocols would take many minutes (maybe even hours) per query, and certainly will not scale to large datasets and long sequences.

In this work we develop an efficient privacy-preserving protocol for computing an approximation of the k-match function from above. Our solution lets the client and server preprocess their respective inputs to the protocol, in a manner that makes the secure computation portion of the work much less expensive. While the preprocessing can include many edit-distance computations, these are all carried out in the clear, and so they are much cheaper than the secure computation portions. Moreover, this preprocessing is reusable and can be used by the server to answer unlimited number of queries (and similarly by the client to query multiple databases).

We show that the implementation of this approximation can handle databases with thousands of records and sequences of length thousands of alleles. We tested our solution on two datasets that were provided by the iDASH organizers (see Section 4). One dataset consisted of 500 sequences, each of length about 3500, that were extracted from human chromosome 3 in a regions that features high divergence among individual genomes (that has variability under 5%). After a one-time preprocessing of around 12 seconds, the server can answer many queries in less than a second each. We then "stress-tested" our implementation using a larger dataset, consisting of 10,000 sequences, which was synthesized by the organizers of the competition and had even more variability than the real one (variability under 10%), and obtained similar results. Below we refer to them as the "real" and "synthesized" datasets. For the sake of comparison, previous works focused on regions with much less divergence (overall variability of 0.5%), and these methods are not sensitive enough to approximate well the distances in regions with such higher diversity.

1.1 Overview of Our Solution

1.1.1 Approximation Function

We develop an efficient approximation algorithm that utilizes the distribution of genomic data. We heuristically expect (and empirically verify) that the our algorithm provides an excellent approximation of the desired functionality.

We first replace the edit-distance function with a block-wise approximation of it. Using a specially tailored method (described below) we break the query Q into n blocks $(Q_1, \ldots, Q_n)$, and similarly break each sequence S_i in the database into blocks $(S_{i,1}, \ldots, S_{i,n})$, where the blocks are very small (typically, no more than 15 letters). Denoting by ED the edit-distance function, we roughly use the approximation

$$\mathsf{ApproxED}(Q, S_i) \approx \sum_{\ell=1}^{n} \mathsf{ED}(Q_\ell, S_{i,\ell}). \tag{1.1}$$

This approximation alone reduces the cost significantly, as computing the distance of $\mathsf{ED}(Q_\ell, S_{i,\ell})$ when $|Q_\ell|, |S_{i,\ell}|$ are very small is cheap, and so the computation of $\mathsf{ApproxED}(Q, S_i)$ can be done in linear time. However, we can still do much better: We observe that for our genomic data, each block position has only a few distinct values (such as $\{\mathtt{TT}, \mathtt{AGT}, \mathtt{AGG}\}$) that actually appear there. Namely, for each $\ell : \ell = 1, ..., n$ the set of values $T_\ell = \{S_{i,\ell} : i = 1, \ldots, m\}$ is much smaller than m. In our tests, even for our largest database with $m = 10,000$, we only had $v = \max_\ell |T_\ell| \leq 40$. This means that the client has to compare each block value with only 40 possible values, and not $10,000$ (saving 99.5% amount of work).

Moreover, in almost all cases ($> 99\%$), the block Q_ℓ was also one of the values in the set T_ℓ. Hence, the server can precompute the values $\mathsf{ED}(u, S_{i,\ell})$ for all $u \in T_\ell$, and computing the approximation above can be done by comparing the client block Q_ℓ to the few values $u \in T_\ell$ (for every $\ell = 1, \ldots, n$).

In more detail, let v be some known bound on the number of distinct values in each block (e.g., $v = 40$), and fix one particular block position $\ell \in \{1, 2, \ldots, n\}$. For every sequence S_i in the database and every value that appears in the ℓ'th block, $u_{\ell,j} \in T_\ell$, the server computes in the clear the edit distance $L_\ell[j, i] = \mathsf{ED}(u_{\ell,j}, S_{i,\ell})$.

Then, for every value $u_{\ell,j} \in T_\ell$, $j = 1, \ldots, v$, the client and server jointly compute the bit $\chi_{\ell,j}$ that indicates whether or not $u_{\ell,j} = Q_\ell$. If the value Q_ℓ happens to be equal to one of the $u_{\ell,j}$'s, then for every $S_{i,\ell}$ we have

$$\mathsf{ED}(Q_\ell, S_{i,\ell}) = \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_\ell[j, i]. \tag{1.2}$$

Namely, in this case we can compute the values that are needed for Eq. (1.1) as a simple linear combination involving the (few) bits $\chi_{\ell,j}$ and the values $L_\ell[j, i]$ that the server knows in the clear. Hence, after computing the v indicator bits $\chi_{\ell,1}, \ldots \chi_{\ell,v}$ for each block position, we approximate the edit distance between Q and S_i using

$$\mathsf{ApproxED}(Q, S_i) = \sum_{\ell=1}^{n} \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_\ell[j, i]. \tag{1.3}$$

We note that in the case where $Q_\ell \notin T_\ell$, the expression on the right-hand side of Eq. (1.2) is always zero, so these cases introduce more error to our approximation. However, we verified

empirically that the effect of this added error is very minor (see Appendix A). Summing up, we compute securely the approximate k-closest-match function, defined as:

$$\begin{aligned}&\mathsf{ApproxClosest}_{k,m}(Q, \{S_1, \ldots, S_m\}) \\ &= \text{indexes } i_1, \ldots, i_k \text{ of the } k \ S_i\text{'s with smallest } \mathsf{ApproxED}(Q, S_i),\end{aligned} \tag{1.4}$$

where ties are broken using the indexes i themselves.

1.1.2 Partitioning into Blocks

A crucial detail of our approximation is the method that we use to partition the sequences S_i and the query Q into blocks. The simplest possibility would be to partition them into fixed-length blocks, but this simplistic partitioning yields a poor approximation. For example, a small shift close to the beginning of the query (perhaps just inserting a single character) can lead to many misalignments in the consecutive blocks, causing this single error to be counted multiple times.

To get a better partitioning method, we utilize specific features of our application domain,[1] specifically the existence of a public "reference genome" R, which is somewhat close to both to the query Q and the database sequences S_i. Our partitioning method begins with applying the simplistic partitioning above to the reference genome R. Then, each party separately aligns its input sequence(s) to the reference sequence (using the Wagner-Fischer algorithm [21]), and we hope that the alignments of Q vs. R and S_i vs. R are "close enough" to induce a good alignment between Q and S_i. We stress that the alignment of Q and the S_i's to R is done locally *and in the clear* by each party. Description of this procedure can be found in Section 3.

1.1.3 The Reference Sequence R

To get a close enough alignment for our partitioning procedure, we need the public reference sequence R to be "somewhat close" to Q and all the S_i's. An obvious choice for this sequence is to extract it from the Standard Reference Assembly GRCh38 [23]. Using this publicly known sequence for partitioning yields very good results for the "real" dataset from the iDASH competition (that has variability under 5%): In our tests, our approximation algorithm returned exactly the k closest sequences in 98% of the runs, and a very good approximation in the remaining 2%, see more details in Section 4.

When working with the "synthesized" dataset (that has about 10% variability), using this standard reference sequence still provided meaningful accuracy, but not quite as good and at a much higher cost. In particular, the number of distinct values per block was much higher. For example with 500-record dataset we had 30 distinct values (vs. only 6 for the "real" dataset), and for the entire 10,000-record synthesized dataset we had up to 178 distinct values in some blocks.

To improve performance and accuracy, we used a *synthetic sequence* that we generated from the dataset itself, as will be described in Section 3.2. Using this synthetic sequence reduces the number of distinct values per block to only 16 for a 500-record dataset and 40 for the full 10,000-record dataset, and also improved accuracy, see the detailed results in Section 4. In the sequel we refer to the sequence extracted from the standard reference assembly GRCh38 as the *global reference*

[1] We remark that using application-specific partitioning method is the best we can hope for: Any general-purpose partitioning that yields linear-time processing (and guarantee accuracy) will violate a conditional lower bound on the complexity of edit-distance calculations [5].

sequence, and the one generated from the database is called the *synthetic reference sequence*. It should be noted that using a sequence that was generated from the database leaks some information about the database, but still guarantees meaningful notion of privacy, see discussion in Section 3.2.

We also describe in Section 3.2 a hybrid procedure with less leakage. In this procedure, the server computes and uses the synthetic reference sequence, but it does not send it to the client. Instead, the client only gets partial information about it, and uses that partial information in conjunction with the global reference sequence to partition its query into blocks. That hybrid procedure achieves performance and accuracy almost as good as using the same synthetic reference by both client and server.

1.1.4 Efficient Secure Computation

Transforming the approximation procedure above into a secure protocol is not a straightforward application of generic transformations (e.g., Yao [24] or GMW [12]). Rather we use the specific form of our approximation to get a significantly faster implementation.

Specifically, we use the fact that Eq. (1.3) is just a matrix-vector multiplication, where the vector of shared bits $\chi_{\ell,j}$ is multiplied by matrix of values $L_\ell[j,i]$ that are known to the server. Once we have an XOR sharing of all the bits $\chi_{\ell,j}$, we can obtain an *additive sharing* of the values $\chi_{\ell,j} \cdot L_\ell[j,i]$ by a simple oblivious transfer protocol (which is of course accelerated using OT-extension [4]), and then the summation is computed locally by the two parties.

In more detail, the protocol begins with a preprocessing phase in which each party is breaking its input into blocks as described above, and the server is computing the sets T_ℓ and all the intra-block edit-distance values $L_\ell[j,i] = \mathsf{ED}(u_{\ell,j}, S_{i,\ell})$.

Next, the parties engage in a standard secure computation protocol for computing a random XOR sharing of all the bits $\chi_{\ell,j}$, using an optimized variant of Yao's garbled circuits. Then for each j, ℓ the parties execute a 1-of-2 oblivious transfer protocol to get a random additive sharing of the value $\chi_{\ell,j} \cdot L_\ell[j,i]$, as described in Section 2.2.

The parties then locally sum up their shares as per Eq. (1.3), thus obtaining an additive sharing of the m approximate edit distance values $\mathsf{ApproxED}(Q, S_i)$. Finally a standard secure computation protocol, using an optimized variant of Yao's garbled circuits, yields the indexes of the k smallest values.

Theorem 1.1 (informal)**.** *The protocols sketched above securely computes the function* $\mathsf{ApproxClosest}_{k,m}$ *from Eq. (1.4) in the semi-honest adversary model.*

1.2 Implementation and Performance

We implemented our protocol using the `C++` version of the Secure Computation API library (SCAPI) [9], and tested it on the 500-sequence "real dataset" provided by the iDASH competition as well as on the "synthesized" dataset where we tested databases of size 500 through 4,000. In our tests, the most costly aspect was the pre-processing on the server side (which is preformed in the clear, and only needed to be done once). We did not optimize this part and it took under 12 seconds for our 500-sequence database (with the length of each sequence $w \approx 3500$). We expect an optimized implementation to be much faster (perhaps by an order of magnitude).

For the online secure computation itself (which is done for every query), the overall number of non-XOR gates is only about 800K AND gates, and we use roughly the same number of OTs. Using efficient implementations of Yao's garbled circuits [19, 25] and OT extensions [4, 18], it took

about 2 seconds to process each query, and less than a second with the real database (that due to its smaller variability allows choice of parameters that mitigate with our protocol.)

1.3 Related Work

Jha et al. proposed in [17] some techniques for secure edit distance using garbled circuits, and shows that the overhead is acceptable only for small strings. (For example, handling 200-character strings takes about 2GB of bandwidth). Using some other optimizations, 500-character string instances take almost an hour to complete. Computing edit distance is also a common benchmark for analyzing improvements in general secure computation techniques and frameworks (see [4, 13, 14], to state a few). These works compute accurate edit distance, and do not utilize the specific input distribution of genomic data.

Recent years saw a large body of work on using secure-computation protocols for genomic data, some surveys include [2,20]. The most relevant prior work to ours is by Wang et al. [22] (building on earlier work of Baldi et al. [6]), that considers the same task: designing a privacy-preserving protocol for supporting the Similar Patient Query. Wang et al. developed an approximation protocol for edit distance that enables computation in a large scale of the *whole* genome (i.e., supporting up to 80M nucleotide, all locations in the genome that are suspected to have variations). The approximation is shown to be very accurate, and the computation is performed in several minutes.

The approach of Wang et al. relies heavily on the fact that the genomes that they are examining have little divergence, i.e. not more than 0.5% distance between genomes and that most differences (80-90%) are substitutions. Using this fact, Wang et al. show how to approximate the edit distance by just considering the set of indexes where the two sequences differ from the reference genome, and running a set-intersection protocol.

The above assumptions are valid in some instances, however, some regions have high divergence and the differences are also caused by insertions and deletions, which are more difficult to deal with in computing edit distance. For example in some regions that affect the immune system the distances between two individuals may be up to 10% (of the size of the region), and about 25% of the differences between two sequences are due to insertions or deletions. Hence the approach of Wang et al. cannot be used and a different method is needed.

In this work we provide a solution that works also for regions with high-divergence and high insertion/deletion rates (as needed for our motivating application of identifying likely conditions based on a small portion of the human genome). We note that the two methods can be combined in applications: Large regions with small variability can be approximated using the more coarse method of Wang et al., while other (smaller) regions with higher variability can be approximated using our more sensitive method.

Security implications of computing an approximation. Feigenbaum et al. observed [10] that computing an approximated version of a function may have security implications, in that the approximated version may leak information which is not revealed by the exact version. This concern certainly applies to our solution. For example it is not hard to see that by engaging in multiple executions with adversarially chosen queries, a client can fully recover the sets of block values T_ℓ from above.

It is interesting to ask to what extent this leakage is problematic in real life applications (and how it can be mitigated), but such questions go beyond the scope of the current work.

Organization. The rest of this paper is organized as follows. We reverse the order of the sections relative to the order in the Introduction and start with the presentation of the secure computation in Section 2. Followed by the description of how to break the sequences into blocks in Section 3, and describe also an algorithm for generating the reference sequence. We report the accuracy of our protocol in Section 4. We conclude with the implementation in Section 5. In the appendixes we report provide some of our experiments, as well as some supplementary data for decision we made in our design.

2 Privacy Preserving Protocol

In this section we present our semi-honest secure protocol for computing the $\mathsf{ApproxClosest}$ function from Eq. (1.4): The client has a query string, the server has a database of records, and the client needs to learn the indices (or labels) of the k closest records to its query, as specified in Functionality 2.1 below.

Functionality 2.1: (Approximate) Closest k Records Functionality

- **Public parameters:** The database size m and output size $k < m$.
- **Private inputs:** The client holds a sequence query Q. The server holds a database DB of m sequences $(S_1, \ldots, S_m)$.
- **The functionality:**
  1. Let $\tilde{e}_i = \mathsf{ApproxED}(Q, S_i)$ be the approximate edit distance between Q and S_i, as per Eq. (2.1).
  2. Let I_k be the set of indexes of the k-smallest values in $\tilde{e}_1, \ldots, \tilde{e}_m$, breaking ties according to the lexicographic order.
- **Output:** The client outputs I_k (ordered lexicographically), the server has no output.

As described in the Introduction we do not compute the exact edit distance between the query and the sequences in the database, but rather an approximation of this value which is amenable to an efficient secure computation. The exact function that we compute depends on our procedure for breaking the sequences and query into blocks, which we describe in detail in Section 3 below. That procedure computes the blocks $Q = (Q_1, \ldots, Q_n)$ and $S_i = (S_{i,1}, \ldots, S_{i,n})$ (where n is a known parameter, in out implementation typically $n = 1200$). For each block location we define a set, T_ℓ, of values that occur in that block position, that is

$$T_\ell = \{S_{1,\ell}, \ldots, S_{m,\ell}\}.$$

The approximate edit distance function that we compute is:

$$\mathsf{ApproxED}(Q, S_i) \;=\; \sum_{\ell=1}^{n} \Delta(Q_\ell, S_{i,\ell}), \text{ where } \Delta(Q_\ell, S_{i,\ell}) = \begin{cases} \mathsf{ED}(Q_\ell, S_{i,\ell}) & \text{if } Q_\ell \in T_\ell \\ 0 & \text{otherwise .} \end{cases} \tag{2.1}$$

We remark that although computing $\mathsf{ED}(Q_\ell, S_{i,\ell})$ also for blocks where $Q_\ell \notin T_\ell$ would improve accuracy, this improvement is minor (since the case of $Q_\ell \notin T_\ell$ is rare). See Appendix A for further discussion of our choice. Our protocol for realizing Functionality 2.1 consists of a local preprocessing stage, followed by two main protocol stages:

Preprocessing: In this stage the parties break their sequences in to blocks, and the server computes several tables. We describe this stage in Section 2.1.

Stage I: computing additive sharing of the approximations. This stage is the crux of our protocol, and is described in Section 2.2. The client and the server interactively compute secret sharing of the vector of approximated edit distances. Specifically, the parties compute additive sharing of the following vector L:

$$L \stackrel{\Delta}{=} (\mathsf{ApproxED}(Q, S_1), \ldots, \mathsf{ApproxED}(Q, S_m)). \tag{2.2}$$

Stage II: computing the k-minimal values. In the second stage of the interaction, the client and the server compute the k minimal values of the secret-shared vector L, and learn the indices of these values. This stage is described in Section 2.3.

2.1 The Preprocessing Stage

The preprocessing stage relies on a procedure $\mathsf{BreakToBlocks}$ that the two parties use to break each of their respective sequences into blocks. That procedure is described in Section 3, and it has the property that for each block location there are only a few distinct values that occur there, and moreover that the two parties know a bound v on the number of values in each block. The $\mathsf{BreakToBlocks}$ procedure is parametrized by public reference sequence R and blocksize parameter b.

The client. On input the query Q, the client sets $(Q_1, ..., Q_n) := \mathsf{BreakToBlocks}_{R,\mathsf{b}}(Q)$.

The server. On input the database, $S_1, ..., S_m$, the server proceeds as follows:

1. Set $(S_{i,1}, ..., S_{i,n}) := \mathsf{BreakToBlocks}_{R,\mathsf{b}}(S_i)$ for all $i = 1, \ldots, m$.
2. For each block location $\ell = 1, ..., n$, compute the set

$$T_\ell = \{S_{i,\ell} : i = 1, \ldots, m\} = \{u_{\ell,1}, \ldots, u_{\ell,v}\} \tag{2.3}$$

   of all the values in the ℓth block. The server pads all sets T_ℓ to be of the same size v using some dummy values.
3. For every block location $\ell = 1, \ldots, n$, every sequence S_i, $i = 1, \ldots, n$, and every value $u_{\ell,j} \in T_\ell$, $j = 1, \ldots, v$, the server computes the edit distance between $u_{\ell,j}$ and $S_{i,\ell}$, setting $L_\ell[j, i] := \mathsf{ED}(u_{\ell,j}, S_{i,\ell})$. Below we denote the row $L_\ell[j, \cdot]$ by $L_{\ell,j}$, namely

$$L_{\ell,j} := (\mathsf{ED}(u_{\ell,j}, S_{1,\ell}), \ldots, (\mathsf{ED}(u_{\ell,j}, S_{m,\ell}))\,. \tag{2.4}$$

   (Jumping ahead, each vector $L_{\ell,j}$ represents the contribution of the ℓ'th block to the final edit distances approximations, for the case where $Q_\ell = u_{\ell,j}$.)

The preprocessing of the server is done only once, and then multiple queries can be computed.

Computing Eq. (2.1)). We observe that for each i, ℓ, the value $\Delta(Q_\ell, S_{i,\ell})$ from Eq. (2.1)) can be expressed as

$$\Delta(Q_\ell, S_{i,\ell}) = \sum_{j=1}^{v} \chi_{\ell,j} \cdot \underbrace{\mathsf{ED}(u_{\ell,j}, S_{i,\ell})}_{=L\ell[j,i]}, \text{ where } \chi_{\ell,j} \triangleq \begin{cases} 1 & \text{if } Q_\ell = u_{\ell,j} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Therefore we have for all i

$$\mathsf{ApproxED}(Q, S_i) = \sum_{\ell=1}^{n} \Delta(Q_\ell, S_{i,\ell}) = \sum_{\ell=1}^{n} \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_\ell[j, i].$$

Thus, the vector of approximations $(\mathsf{ApproxED}(Q, S_i))_i$ can be computed as

$$L = \big(\mathsf{ApproxED}(Q, S_1), \ldots, \mathsf{ApproxED}(Q, S_m)\big) \;=\; \sum_{\ell=1}^{n} \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_{\ell,j} \;. \tag{2.6}$$

2.2 Stage I: Computing Additive Sharing of the Vector L

After preprocessing, the client holds a vector of blocks $(Q_1, ..., Q_n)$, and the server holds all the (ordered) sets $T_1, \ldots, T_n$ and the edit-distance vectors $L_{\ell,j}$ for $\ell = 1, \ldots, n$ and $j = 1, \ldots, v$. Our goal in the first stage of interaction is to compute additive sharing of the approximate-distance vector L. Formally, we need to realize the Functionality 2.2 below:

Functionality 2.2:
Additive Sharing of Approximate Edit-Distances, $L^c - L^s = L$

- **Parameters**: Let d be a public upper bound on $\max_{i \in m} \mathsf{ApproxED}(Q, S_i)$.
- **Input:** The client inputs the blocks $(Q_1, \ldots, Q_n)$. The server inputs the tables $T_\ell = \{u_{\ell,1}, \ldots, u_{\ell,v}\}_{\ell \in [n]}$ and vectors $\{L_{\ell,j}\}_{\ell \in [n], j \in [v]}$.
- **The functionality:**
  1. Let $L = \sum_{\ell=1}^{n} \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_{\ell,j} \in d^m$
     ($L_{\ell,j}$, $\chi_{\ell,j}$ are defined in Eq. (2.4), Eq. (2.5), respectively);
  2. Choose a random vector $L^s \in d^m$ and set $L^c := L + L^s \bmod d$.
- **Output:** The client outputs L^c while the server outputs L^s.

The protocol for realizing Functionality 2.2 consists of two main steps:

- First, the parties compute shares of the indicator bits $\chi_{\ell,j}$. That is, for every $\ell \in [n], j \in [v]$, the client and server receive random bits $\chi^c_{\ell,j}$, $\chi^s_{\ell,j}$, respectively, s.t. $\chi^c_{\ell,j} \oplus \chi^s_{\ell,j} = \chi_{\ell,j}$.
- Next they use oblivious transfer to convert their shares of $\chi_{\ell,j}$ (and the value $L_{\ell,j}$ held by the server) into additive shares of $\chi_{\ell,j} \cdot L_{\ell,j}$. That is, they choose random vectors $L^c_{\ell,j}, L^s_{\ell,j}$ s.t. $L^c_{\ell,j} - L^s_{\ell,j} = \chi_{\ell,j} \cdot L_{\ell,j} \pmod d$.

Then the client and server locally sum their shares: The client is computing $L^c = \sum_{\ell=1}^{m} \sum_{j=1}^{v} L^c_{\ell,j} \bmod d$, and the server $L^s = \sum_{\ell=1}^{m} \sum_{j=1}^{v} L^s_{\ell,j} \bmod d$. Hence

$$L^c - L^s = \sum_{\ell=1}^{n} \sum_{j=1}^{v} L^c_{\ell,j} - L^s_{\ell,j} = \sum_{\ell=1}^{n} \sum_{j=1}^{v} \chi_{\ell,j} \cdot L_{\ell,j} = L \pmod d.$$

2.2.1 Step 1: Indicator Bits

This step realizes the Functionality 2.3:

Functionality 2.3:
Computing XOR sharing for the indicator bit ($\chi^c_{\ell,j} \oplus \chi^s_{\ell,j} = \chi_{\ell,j}$)

- **Input:** The client inputs the block Q_ℓ.
  The server inputs $u_{\ell,j}$, which is the jth value in the set T_ℓ.
- **The functionality:** Let $\chi_{\ell,j} = 1$ if $Q_\ell = u_{\ell,j}$, and $\chi_{\ell,j} = 0$ otherwise. Choose a random bit $\chi^s_{\ell,j}$ and set $\chi^c_{\ell,j} = \chi^s_{\ell,j} \oplus \chi_{\ell,j}$.
- **Output:** The client outputs $\chi^c_{\ell,j}$ while the server outputs $\chi^s_{\ell,j}$.

We realize Functionality 2.3 using a direct application of Yao's protocol. Let $Q_\ell = \sigma_1, \ldots, \sigma_t$ and $u_{\ell,j} = \tau_1, \ldots, \tau_t$, representing the inputs $Q_\ell, u_{\ell,j}$ (each padded to some bound b' and converted to binary using suffix-free encoding).[2] The server chooses a random bit $\chi^s_{\ell,j}$ (which will also be its output of the protocol), and we use a standard secure protocol (e.g., Yao's protocol) in which the client learns the output bit $\chi^c_{\ell,j} = \chi^s_{\ell,j} \oplus \bigwedge_{k=1}^{t}(\sigma_k \oplus \tau_k \oplus 1)$. At the end of this stage, the client and the server holding bits $\chi^c_{\ell,j}, \chi^s_{\ell,j}$ (resp.) for every $\ell = 1, \ldots, m$ and $j = 1, \ldots, v$.

2.2.2 Step 2: Additive sharing

This step realizes the Functionality 2.4.

Functionality 2.4: Computing additive sharing for $\chi_{\ell,j} \cdot L_{\ell,j}$

- **Parameters**: The edit-distance bound d.
- **Input:** Client inputs is $\chi^c_{\ell,j}$, server inputs is $\chi^s_{\ell,j}$ and the vector $L_{\ell,j}$.
- **The functionality:** Set $\chi_{\ell,j} = \chi^c_{\ell,j} \oplus \chi^s_{\ell,j}$. Choose a random vector $L^s_{\ell,j} \in d^m$ and set $L^c_{\ell,j} = L^s_{\ell,j} + \chi_{\ell,j} \cdot L_{\ell,j}$.
- **Output:** The client outputs $L^c_{\ell,j}$ and the server outputs $L^s_{\ell,j}$.

We realize Functionality 2.4 using oblivious transfer, as described in Protocol 2.5.

2.2.3 Putting it together: realizing Functionality 2.2

We realize functionality 2.2 in Protocol 2.6 using functionalities 2.3 and 2.4.

2.3 Stage II: Finding k Minimal Values

After computing an additive sharing of the approximate edit distances between Q and the m records $S_1, \ldots, S_m$, the parties engage in a protocol to find the k smallest distances. The full specification is found in Functionality 2.7.

The protocol to realize Functionality 2.7 is just a direct application of Yao's protocol, applied to the simple circuit that computes $L = (L_1, \ldots, L_m) = L^c - L^s \bmod d$, then repeatedly finds the

[2] In our case the original strings were over a 4-ary alphabet, so to get suffix-free encoding we need to set at least $t = 2\mathsf{b}' + 1$.

Protocol 2.5: Realizing Functionality 2.4 (in the OT-hybrid model)

- **Parameters**: The edit-distance bound d.
- **Input:** Client inputs is $\chi^c_{\ell,j}$, server inputs is $\chi^s_{\ell,j}$ and the vector $L_{\ell,j}$.
- **The protocol:** (all additions are done (mod d))
  1. The server chooses a random vector $L^0_{\ell,j}$ and sets $L^1_{\ell,j} = L^0_{\ell,j} + L_{\ell,j}$.
  2. The server and the client engage in a 1-out-of-2 oblivious transfer. The client as the receiver with the choice bit $\chi^c_{\ell,j}$, and the server as the sender with inputs:
     - $(L^0_{\ell,j}, L^1_{\ell,j}) = (L^0_{\ell,j}, L^0_{\ell,j}) + (0, L_{\ell,j})$ in case $\chi^s_{\ell,j} = 0$,
     - $(L^1_{\ell,j}, L^0_{\ell,j}) = (L^0_{\ell,j}, L^0_{\ell,j}) + (L_{\ell,j}, 0)$ in case $\chi^s_{\ell,j} = 1$.

     Let $L^c_{\ell,j}$ denote the output that the client receives from the OT protocol.
- **Output:** The server outputs $L^s_{\ell,j} = L^0_{\ell,j}$ and the client outputs $L^c_{\ell,j}$.

Protocol 2.6: Realizing Functionality 2.2

- **Parameters**: Let d be a public upper bound on $\max_{i \in m} \mathsf{ApproxED}(Q, S_i)$.
- **Input:** The client inputs the blocks $(Q_1, \ldots, Q_n)$. The server inputs the tables $T_\ell = \{u_{\ell,1}, \ldots, u_{\ell,v}\}_{\ell \in [n]}$ and vectors $\{L_{\ell,j}\}_{\ell \in [n], j \in [v]}$.
- **The protocol:** (all additions are done mod d)
  1. For every $\ell = 1, \ldots, n$ and $j = 1, \ldots, v$:
     (a) Invoke Functionality 2.3, with client input Q_ℓ and server input $u_{\ell,j}$. Let $\chi^c_{\ell,j}$, $\chi^s_{\ell,j}$ be the outputs of the client and server, respectively.
     (b) Invoke Functionality 2.4 with client input $\chi^c_{\ell,j}$ and server input the bit $\chi^s_{\ell,j}$ and the vector $L_{\ell,j}$. Let $L^c_{\ell,j}, L^s_{\ell,j}$ be the output of the client and server, respectively.
  2. The client computes $L^c = \sum_{\ell=1}^{n} \sum_{j=1}^{v} L^c_{\ell,j}$,
     the server computes $L^s = \sum_{\ell=1}^{n} \sum_{j=1}^{v} L^s_{\ell,j}$.
- **Output:** The client outputs L^c, the server outputs L^s.

minimum k times (breaking ties using the index).[3]

Realizing Functionality 2.1 is now a straightforward application of the components above, as summarized in Protocol 2.8 below.

2.4 Security Analysis

Below we sketch the security analysis of our protocol. We follow the standard definition of static semi-honest security in the standalone model (cf. [11]). Namely we need to describe a simulator for the two cases of corrupted client and corrupted server, that gets the input and output of the corrupted party and needs to generate a protocol view that is indistinguishable from the view of the real protocol. For randomized functionality we consider the joint distribution of the view of the corrupted party and the output of all parties. We argue the security in a bottom-up fashion:

- We assume semi-honest security of the basic building blocks that we use, i.e., oblivious-transfer

[3] Breaking ties according to index is easy when using a comparison tree with the leaves ordered by their index, you just always break ties in favor of the left child.

Functionality 2.7: Find the k-Minimal Values

- **Parameters:** number of records m, output size k, distance bound d.
- **Input:** Client and server hold $L^c, L^s \in \mathbb{Z}_d^m$, respectively.
- **The functionality:**
  1. Let $L = L^c - L^s \bmod d$, and denote $L = (L_1, \ldots, L_m)$
  2. Find the k smallest values in the sequence L, using the indexes $1, \ldots, m$ to break ties.

  Output: The client gets m bits $(\sigma_1, \ldots, \sigma_m)$, where $\sigma_j = 1$ if $L[j]$ is one of the k smallest values. The server has no output.

Protocol 2.8: Realizing Functionality 2.1

- **Parameters:** Database size m, output size $k < m$, distance bound d.
- **Input:** The client holds a sequence query Q. The server holds a database DB of m sequences $(S_1, \ldots, S_m)$.
- **The protocol:**
  1. The clients and the server perform the preprocessing stage. The client holds the blocks $Q_1, \ldots, Q_n$, and the server holds the tables $T_1, \ldots, T_\ell$ and the vector $\{L_{\ell,j}\}_{\ell \in [n], j \in [v]}$.
  2. The parties invoke Functionality 2.2, where the client inputs $Q_1, \ldots, Q_n$ and the server inputs the vector $\{L_{\ell,j}\}_{\ell \in [n], j \in [v]}$ and the tables $\{T_\ell\}_{\ell=1}^n$.
     The client receives vector L^c and the server receives the vector L^s.
  3. The parties invoke Functionality 2.7, where the client inputs L^c and the server inputs L^s. The client receives the m bits $(\sigma_1, \ldots, \sigma_m)$.
- **Output:** The client outputs $(\sigma_1, \ldots, \sigma_m)$.

and Yao's protocol.

- This immediately implies the security for the find-k-min protocol for realizing Functionality 2.7, as well as the security of the protocol for realizing Functionality 2.3 (as both of these are direct applications of Yao's protocol).
- Next we prove the security of Protocol 2.5 that realizes Functionality 2.4 (sharing of $\chi_{\ell,j} \cdot L_{\ell,j}$), and then Protocol 2.6 that realizes Functionality 2.2 (sharing of L).
- Finally we put everything together and prove security of the entire protocol (Protocol 2.8).

Note that only the last two bullets require new proofs, everything else holds by assumption on the components that we use.

Theorem 2.9 (Sharing of $\chi_{\ell,j} \cdot L_{\ell,j}$)**.** *Assuming the semi-honest security of the underlying* 1*-out-of-*2 *OT protocol, the Protocol 2.5 securely realizes Functionality 2.4 against static corruptions in the semi-honest adversary model.*

Proof Sketch: Correctness if easy to see, just by inspecting the output in the four cases $(\chi^c_{\ell,j}, \chi^s_{\ell,j}) \in \{(0,0), (0,1), (1,0), (1,1)\}$.

Regarding privacy, as the protocol is just an execution of an OT protocol, the view of the server is its random tape, $L^s_{\ell,j}$. Thus, in order to simulate a corrupted server, the simulator, upon

receiving the output $L^s_{\ell,j}$ from the functionality, just outputs this vector. In case of a corrupted client, the view of the client is just its final output. The theorem follows. ■

Theorem 2.10 (Sharing of L)**.** *Protocol 2.6 securely realizes Functionality 2.2 against static corruptions in the semi-honest adversary model.*

Proof: Correctness is easy by inspection. As for security, assume the case of a corrupted client. The simulator receives a random L^c as the output of the corrupted client, and the honest server receives $L^s = L^c - L$. The view of the client during the execution is the set of shares $\{\chi^c_{\ell,j}\}_{\ell,j}$ and the vectors $\{L^c_{\ell,j}\}_{\ell,j}$. The simulator just chooses the set of bits $\{\chi^c_{\ell,j}\}_{\ell,j}$ uniformly at random, and also chooses the vectors $\{L^c_{\ell,j}\}_{\ell,j}$ at random from $d^{n \cdot v}$ under the constraint that $\sum_{\ell=1}^{n}\sum_{j=1}^{v} L^c_{\ell,j} = L^c$, and outputs these values. As the intermediate values $\{\chi^s_{\ell,j}\}_{\ell,j}$ are hidden from the distinguisher, the bits $\{\chi^c_{\ell,j}\}_{\ell,j}$ that the client receives from the invocations of Functionality 2.4 are distributed uniformly. Moreover, as the values $\{L^s_{\ell,j}\}_{\ell,j}$ are also hidden from the distinguisher, the vectors $\{L^c_{\ell,j}\}_{\ell,j}$ are all random under the constraint that they sum-up to the output of the client. The case of a corrupted server is proven analogously. ■

Theorem 2.11 (Overall protocol)**.** *Protocol 2.8 realizes Functionality 2.1 against static corruptions in the semi-honest adversary model.*

Proof Sketch: As before, correctness is easy, and we need to show privacy. We note that except for the input and output values, the only thing beyond that the parties see in Protocol 2.8 are the vectors L^c, L^s that are returned by the intermediate Functionality 2.2, and that these vectors are individually uniform, irrespective of the input and output. Thus, in the case of a corrupted client the simulator just chooses L^c uniformly at random, and in the case of a corrupted server it chooses L^s uniformly at random. ■

3 Breaking Sequences into Blocks

As stated in the Introduction the main idea underlying our solution is to approximate the edit distance between two sequences S and Q by partitioning both sequences into n blocks each, then summing up the edit distances across all blocks, returning $\sum_{\ell=1}^{n} \mathsf{ED}(Q_\ell, S_\ell)$. In this section we describe the method that we use to partition the sequences into blocks.

The idea of approximating the edit distance by computing the edit distance on small blocks is appealing as it yields an extremely efficient secure computation. However, the simplest manner of breaking the sequence/query into equal size blocks did not yield a good approximation of the edit distance over the full sequence. Thus, the question arose whether we can enable both parties to break their sequences into blocks that would also yield a good approximation of the edit distance.

3.1 Utilizing a Reference Genome

In order to refine the breaking into the blocks we introduced a commonly known reference genome R and had both the server and the client break their sequences in relation to R. The anticipation was that if we chose the reference genome R wisely, then it would aid in breaking the sequence and the query into blocks in a manner that would yield a better approximation of the full edit distance. The breaking of the sequences/query is done by computing an edit distance between the

sequence/query and the reference genome. It might seem that we have accomplished nothing as we require a full computation of edit distance. However, the fact of the matter is that we off-load this computation to the server and client to be computed locally. The local computations are done in the clear and thus are much more efficient than computing them via a secure computation. Moreover, as we have seen, the preprocessing is re-usable, and for a large amount of queries this overhead becomes minor.

In our solutions we rely on a reference genome R of roughly the same length as the sequences that we want to break. To break a sequence S (or a query Q) into blocks, we run the Wagner-Fischer edit distance algorithm (for full details see Section C) to compute the edit distance between R and S. The algorithm also returns the PTR matrix that keeps the alignment between R and S. From the upper-left corner of the PTR to the lower-right corner it traces the path of how the minimum edit-distance can be obtained.

Let b be a parameter representing our desired block size (on the selection of b see Section 5). With S being recorded at the top of the matrix we break it as follows. We traverse the minimum edit distance path in PTR and whenever we have moved down b rows we break the sequence in that position into a block. Note, that the sizes of the blocks in this partition will vary. Most blocks are of size b, but some are shorter or longer. A full specification of the partitioning algorithm can be found in Algorithm 3.1.

Algorithm 3.1: $\mathsf{BreakToBlocks}_{R,\mathsf{b}}(S)$ **– Partition a Sequence into Blocks**

- **Parameters:** A reference sequence $R = (\rho_1, \ldots, \rho_r)$, block-size parameter b.
- **Input:** A sequence $S = (\sigma_1, \ldots, \sigma_s)$
  1. Invoke $\mathsf{ED}(R, S)$, and store the table PTR.
  2. Start at the top-left corner of PTR for each multiple of b, i.e. $\mathsf{b}, 2\mathsf{b}, \ldots$ find the index $j_1, j_2, \ldots$ such that $(i\mathsf{b}, j_i)$ is on the minimum edit-distance path. (If there is more than one pair for the same value $i \cdot \mathsf{b}$, store the index j that is closest to $i \cdot \mathsf{b}$.)
  3. Denote $n = \lceil r/\mathsf{b} \rceil$ (observe that the previous steps defines exactly $n-1$ indexes). Let $j_1, \ldots, j_{n-1}$ be the stored indexes, set $j_0 = 0$ and $j_n = s$. Define the blocks $S_\ell = (\sigma_{1+j_{\ell-1}}, \ldots, \sigma_{j_\ell})$ for every $1 \leq \ell \leq n$.
- **Output:** Output the blocks $S_1, \ldots, S_n$.

Experimental observations. A major factor in both accuracy and the performance is the number of distinct values for each block (i.e., $|Q_\ell|$). If the number of distinct blocks is small for large b, then we can save a lot in running time, as n (the number of blocks) is smaller, while still the work that we have to spend per block is relatively small.

The maximal number of different values in each block of course grows with the block-size parameter b, but it always remains much smaller than the number of records in the database. In our tests on a dataset of 500 records taken from coding region of gene ZNF717 (see Section 4 about our datasets), we never saw more than 10 distinct values in a block, even when the blocksize parameter was as large as $\mathsf{b} = 12$. In a different, synthesized, dataset, we had some more values but still much less than the number of records. (In this synthesized dataset the number of distinct values in each block in our experiments seemed to grow linearly, and was always less than $7 \cdot \mathsf{b}$.)

3.2 Using a Synthetic Reference Sequence

As we mentioned above, using the global reference genome for the BreakToBlocks procedure on our synthesized dataset results in more distinct values per block relative to the real database (up to several hundreds in the maximal DB, see Figure 2 in the appendix). This, degrades both efficiency and accuracy of our protocol. We discovered that the main reason for that is the distance between the reference genome and the database: While the average distance between the real database and the reference genome is 140, in the synthesized database the average was close to 300.

We therefore generate a reference sequence which is closer to the database, by deriving it from the database itself. We remark that the global reference genome was also generated from a database of sequences[4]. Our generation procedure is as follows. We start by breaking all the sequences in the database into block using the global reference genome, then in each block we take the plurality value in that block, and concatenate all these block values.

Algorithm 3.2: Generating Optimized Reference Genome

- **Public parameters:** Block size b and reference sequence $R = (\rho_1, \ldots, \rho_r)$. Let $n = \lceil r/\mathsf{b} \rceil$.
- **Inputs:** DB of m sequences $(S_1, \ldots, S_m)$.
  1. Invoke $(S_{j,1}, \ldots, S_{j,n}) = \mathsf{BreakToBlocks}_{R,\mathsf{b}}(S_j)$ for $j = 1, \ldots, m$.
  2. For every block $\ell = 1, \ldots, n$, let R'_ℓ be the plurality value among the ℓth block values of all sequences, i.e., among $(S_{1,\ell}, \ldots, S_{m,\ell})$.

  Output: Output the sequence $(R'_1, \ldots, R'_n)$.

We then use R' (rather than R) in order to break the sequences S_j and Q into blocks in our approximate edit distance algorithm. We stress that while the server can compute R' from its own input, the client cannot, hence the value of R' leaks to the client some information about the database. Specifically, it leaks the most common value for every block. We remark, however, that in almost all cases the most common value appears many more times than any other values, so the value R' is not very sensitive to the inclusion or absence of any single sequence S_j. In this view, the leaked information still maintain some measure of privacy for the individual sequence in DB. See Appendix B for further discussion on the privacy of this approach and for the empirical evaluation.

Using the synthetic reference genome significantly reduces the size of the tables in the synthesized database (see Figure 2 in the appendix), accelerating the running time of our protocol.

Reducing the leakage. We can reduce the leakage even further, without reducing the accuracy or increasing the online running time at all, using a hybrid approach. As before, we let the server derive the synthetic reference string R' from its database, and use R' to break its sequences into blocks. However, then we use R' to break the *global reference* R into blocks, namely running $(R_1, \ldots, R_n) := \mathsf{BreakToBlocks}_{R',\mathsf{b}}(R)$, and send the breakpoints of R (i.e., the indexes $(j_1, \ldots, j_n)$ in Algorithm 3.1) all the blocks begin in R), instead of sending R'. The client then uses R rather than R' to break its query into block, but it uses a variant of the BreakToBlocks procedure in which the reference string is broken as $(j_1, \ldots, j_n)$ and not on b-block boundaries $(\mathsf{b}, 2\mathsf{b}, \ldots, n\mathsf{b})$.

We note that since the server just uses the synthetic reference string then we get a small number of distinct values per block just as before, so the protocol is as fast as it was. The only thing which

[4]For instance, the global reference genome GRCh37 was derived from 13 anonymous volunteers from Buffalo, New York [1].

is affected is accuracy, and our tests conclude that it does not degrade much. (For example, with our setting of $\mathsf{b} = 3$ the distance between the query and the answers returned by the algorithm increase from an average of +0.34 above the accurate answer to +0.56 (and the rank of the returned results only increases by about one)).

4 Accuracy of Our Approximation

We specifically target "high-divergence" regions of the genome, so we seek to verify that we still get good results even for such regions. We tested our approach on two different datasets, both provided by the organizers of the iDASH competition [15]:

- The first dataset consists of 500 gene sequences extracted from the publicly available 1000 Genomes Project [16]. It was extracted from human chromosome 3 (75785026-75788496), of length just under 3500, within the coding region of gene ZNF717. The iDASH organizers explained the choice of this particular gene by its high divergence among individual genomes.

- A larger dataset of 10,000 sequences was generated via simulation, and has even a significantly higher divergence than the "real" dataset. We used this synthesized dataset to "stress test" our algorithm, often choosing at random just 500 of these 10,000 sequences, so that we can compare the results against the 500-record "real" dataset.

  In addition to being "extremely divergent," this synthesized dataset is quite far from the public reference genome, the average edit distance between a sequence in this dataset and the global reference string was close to 300 (vs. about 140 for the "real" dataset). This made our approach somewhat less accurate and also slower, since we encounter many possible values in each block. We therefore used the synthetic reference sequence for this database (see Section 3.2).

The main results of our accuracy test are summarized Tables 1 and 2. Our algorithm performed remarkably well on the "real" dataset, where it returned the exact correct answer in 98% of the runs, and in the other 2% it returned a sequence with edit distance one larger than the correct answer. Moreover, for this data we has very small number of values in each each block (no more than 10 values), so we chose v to be small and thus the computation was very fast (less than a second per query). For the synthesized dataset we mostly used the synthetic reference genome, and still got good accuracy even though the data is much more divergent: it returned the exact correct answer in 70% of the runs, and in 99% it returned a sequence with edit distance at most one larger than the correct answer. The tables (i.e., $|Q_\ell|$) here were also a little larger, but still small enough to get a fast protocol. Specifically for our setting of $\mathsf{b} = 3$ we had at most 18 distinct values in each block of every run.

Table 1 summarizes the performance and accuracy results of our approximation algorithm as a function of the blocksize parameter b, for our "real" and "synthetic" size-500 datasets, when returning the closests 5 (approximate) distances. The "real" data was using the global reference sequence, while the synthetic one uses mostly the synthetic reference. The table consists of the following columns:

- $\underline{\mathsf{b}'}$ is the largest actual blocksize obtained for any of the sequences (i.e., after breaking the sequences into blocks, some blocks can be larger than b.)

500-record dataset taken from gene ZNF717, global reference genome.
Average(stdev) edit-distance between query and 5'th closest sequence is 4.15(8.37).

Block Size (b)	b'	# Values	Average-Δ (std)	Average Rank (std)	Max-K
3	10	6	0 (0)	5 (0)	6
5	12	8	0.01 (0.1)	5.01 (0.1)	9
8	15	10	0 (0)	5 (0)	7
12	19	10	0.01 (0.1)	5.01 (0.14)	9

500-record synthetic dataset, using global/hybrid reference sequence, with $\mathsf{b} = 3$.
Average(stdev) edit-distance between query and 5'th closest sequence is 139.08(27.39).

Reference Seq.	b'	# Values	Average-Δ (std)	Average Rank (std)	Max-K
Global	10	41	0.98 (1.63)	7.24 (5.21)	45
Hybrid	4	18	0.53 (0.61)	5.95 (1.45)	16

Same 500-record synthetic dataset, using a synthetic reference sequence

Block Size (b)	b'	# Values	Average-Δ (std)	Average Rank (std)	Max-K
3	4	18	0.34 (0.51)	5.47 (0.78)	10
5	6	32	0.34 (0.51)	5.63 (1.29)	14
8	9	54	0.75 (0.90)	6.49 (2.19)	17
12	13	75	1.37 (1.19)	7.89 (3.33)	24

Table 1: Algorithm accuracy as function of block size b, for two 500-record datasets (with $k = 5$).

- # Values is the largest number of distinct values found in any block (i.e., the parameter v).
- Average-Δ is how much farther is the farthest record returned by the algorithm than the true k'th-closest record.
- Average-rank is the true rank of the farthest result returned.
- Max-K is how many records we should have returned to ensure that the true k'th closest record is always part of the result set.

In this experiment, we chose $\approx 1\% \approx 5$ sequences as query sequences, and the other records were chosen to be the database. We repeated this choice 100 times, and the "Average-$\star$" values are computed over these 100 runs, together with the standard deviation (in parenthesis).

Table 2 describe the accuracy of our algorithm as a function of the database size (DB Size), for the synthetic dataset, blocksize parameter $\mathsf{b} = 3$, and when returning $k = 5$ records. The reference genome R is the synthesized reference genome. Similar results are obtained for other block sizes as well. The table consists of the following columns:

- $\%Q$ the distribution of the true rank of the k'th element in the result returned. For all DB size, at least 96% of the queries have rank at most 8.
- ΔED is the average of how much farther the farthest recored returned by the algorithm than

DB Size		Max-rank of any returned record 5	6	7	8	≥ 9	MaxK (Rank)	Real ED (std)
500	%Q	68	20	9	3	0	10 (8)	139.08 (27.39)
	ΔED	0(0)	1.00(0)	1.11(0.56)	1(0)	0 (0)		
1000	%Q	72	16	5	4	3	19 (18)	136.29 (27.25)
	ΔED	0(0)	1.06(0.24)	1(0)	1.5(0.5)	2.33 (0.47)		
2000	%Q	65	22	9	2	2	20 (12)	135.49 (27.19)
	ΔED	0(0)	1.05(0.2)	1.11(0.32)	1(0)	1.5 (0.5)		
4000	%Q	67	18	10	2	3	14 (11)	134.47 (27.2)
	ΔED	0(0)	1(0)	1.10(0.3)	1(0)	1 (0)		
8000	%Q	76	10	6	4	4	12 (10)	133.57 (27.15)
	ΔED	0(0)	1(0)	1(0)	1.25(0.43)	1.25 (0.43)		

Table 2: Accuracy of our algorithm, for varying DB sizes. Each row represents the results of our algorithm with synthetic reference genome, $\mathsf{b} = 3$ and $k = 5$.

the true k'th-closest record. The average is taken among the queries of the corresponding rank $r \in \{5, \ldots, 8\}$ or ≥ 9. The value in parenthesis is the standard deviation.

- Max-K is how many records we should have returned to ensure that the true k'th closest record is always part of the result set. The (Rank) values in parenthesis are the maximum true rank of any record in the result set.
- Real ED is the average (over all queries) of the accurate edit-distance between the query and the true k'th-closest database record.

5 Evaluation and Performance

The "real" dataset for the iDASH competition has sequences that are fairly close together and also close to the reference genome. Specifically, we had a database with $m = 500$ records of size less than $w = 3500$, and edit distances below $d = 256$, but the closest five to every query were always within edit distance of less than 50. Also we kept the block size parameter at a low $\mathsf{b} = 3$, and as a result the sizes of blocks in all our sequences were always of length below 4 (but we used a larger $\mathsf{b}' = 16$ to ensure that there are no errors) and the number of distinct values per block was bounded below 8 (but here too we used $v = 16$ to ensure that there are no errors).

To stress-test our approach we did most of our testing on synthetic data that shows larger variability. We received from the iDASH organizers 10,000 synthetic sequences, we chose 100 of them to be query sequences and $m = 500, 1000, 2000$ or 4000 of them[5] to form databases of difference sizes, and generated a synthetic reference string as explained in Section 3.2. In all these cases, the

[5]Unlike accuracy, in our secure protocol we stopped with a database of size 4000 due to some technical problems with our implementation, and in principal the running times should scale similarly to larger datasets. We remark that our protocol implementation was also tested and evaluated by external referees as part of our participation in the iDash competition, and similar running times were confirmed.

DB Size	$v =$ # values	Preprocessing (s)	Query			Bandwidth (MB)	# AND-gates
			Compare (s)	OTs (s)	k-min (s)		
500	15	14.9	0.9	1.4	0.05	80	789595
1000	25	30	1.51	4.36	0.16	180	1399480
2000	30	61.8	2.1	11.7	0.31	340	2035415
4000	35	119	2.8	28.2	0.6	660	3149350

Table 3: Running times for varying DBs. In all executions, we also produce a synthetic reference genome, which takes 7.6 seconds to produce, and should be added to the preprocessing time.

block-sizes that we obtained were bounded by 4 and the number of distinct values per block was at most 35 even for the largest database (here we used the bounds $\mathsf{b}' = 16$ and $v = 35$ in the protocol).

We implemented our protocol over the `C++` version of the Secure Computation API library (SCAPI) [9]. We use the state-of-the-art improvements, include Yao with free-XOR technique [19] and half-gates [25], and the recent improvements in OT-extension [4, 18]. Table 3 presents the performance results for varying database size, these numbers were obtained by running the protocol on a single x86_64 machine using the loopback device for client-server communication.

In our implementation, the most costly aspect was the pre-processing on the server side (which only needs to be done once per database). This part requires many edit distance computations (in the clear), and we did not attempt to optimize it. In the online time, the most expensive part are the OT protocols.

We report the maximal allowed size of the tables (# values, i.e., v in the protocol), the times took for the preprocessing, answering a query, the bandwidth and the number of AND-gates.

On the choice of parameter b. In terms of accuracy, we observe similar results for different block sizes. Nevertheless, in terms of efficiency, it seems more appealing to run with larger block sizes b. Specifically, recall that the running time is proportional to $n \cdot v$, where n is the number of blocks and v is the size of $|T_\ell|$ (recall that this is the number of indicator bits that we compute, as well as the number of strings $L_{\ell,j}$ that the parties share). Thus, larger block size b implies smaller n.

This is true as long as the size of the tables v remains small. For the real database, v is small even for $\mathsf{b} = 12$, on the other hand, for the synthesized database, we run mainly with $\mathsf{b} = 3$. Of course, the server can locally compute the parameter v for several different block sizes b, and chose the one that minimizes $n \cdot v$ (and sends the client the parameters b and v at the beginning of the protocol).

6 Conclusions

In this work we described a privacy preserving protocol for answering Similar Patient Queries (SPQ) on genome data. Our protocol was designed to operate in settings with high divergence between individuals, where a previous solution due to Wang et al. [22] does not apply. We developed an efficient method for approximating the edit distance that provides very good accuracy even in regions of the genome with 5-10% variability, while at the same time being 2-3 orders of magnitude faster than exact calculation.

Our work was motivated by the 2016 iDASH competition for competing on genome data (and our solution won that competition by being both the most accurate and the fastest among all the submitted entries). In particular, for the 500-record dataset used in that competition, we can answer SPQ in under two seconds per query (after about 15 seconds of one-time pre-processing of the database). We believe that this solution is applicable in real-life situations where SPQ on remote genome data is desirable, and plan to peruse real-world uses of it in the future.

Acknowledgment

We thank Shalev Keren, Meital Levy and Assi Barak for the implementation of our protocol. We also thank Haixu Tang, XiaoFeng Wang, Shuang Wang, Xiaoqian Jiang, Lei Wang, and Diyue Bu for organizing the iDASH competition and helping us with all our questions and requests.

References

[1] E pluribus unum. *Nature Methods*, 7(5):331–331, 05 2010.

[2] M. Akgün, A. O. Bayrak, B. Ozer, and M. Şamil Sağıroğlu. Privacy preserving processing of genomic data: A survey. *Journal of Biomedical Informatics*, 56:103 – 111, 2015.

[3] A. Andoni and K. Onak. Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648, 2012.

[4] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM Conference on Computer and Communications Security*, pages 535–548. ACM, 2013.

[5] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 51–58, 2015.

[6] P. Baldi, R. Baronio, E. D. Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 691–702, 2011.

[7] I. Dinur and K. Nissim. Revealing information while preserving privacy. In F. Neven, C. Beeri, and T. Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 202–210. ACM, 2003.

[8] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 265–284, 2006.

[9] Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012. A link to the library: `http://crypto.biu.ac.il/about-scapi`.

[10] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. N. Wright. Secure multiparty computation of approximations. In *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 927–938. Springer, 2001.

[11] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM Symposium on Theory of Computing, STOC*, pages 218–229, 1987.

[13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

[14] Y. Huang, C. Shen, D. Evans, J. Katz, and A. Shelat. Efficient secure computation with garbled circuits. In *Information Systems Security - 7th International Conference, ICISS 2011, Kolkata, India, December 15-19, 2011, Procedings*, pages 28–48, 2011.

[15] iDASH - integrating Data for Analysis, Anonimization, and SHaring. Webpage at `https://idash.ucsd.edu/genomics`, 2016 competition at `http://www.humangenomeprivacy.org/2016/`.

[16] International Genome Sample Resource. IGSR and the 1000 genomes project. `http://www.internationalgenome.org/`, Accessed Nov 2016.

[17] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 216–230, 2008.

[18] M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology - CRYPTO*, pages 724–741, 2015.

[19] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP*, pages 486–498, 2008.

[20] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. A. Malin, and X. Wang. Privacy in the genomic era. *ACM Computing Surveys (CSUR)*, 2015.

[21] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1), Jan. 1974.

[22] X. S. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 492–503, New York, NY, USA, 2015. ACM.

[23] Wikipedia. Reference genome — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Reference_genome`, 2017. [Online; accessed 10-February-2017].

Frequency of number of block of the queries Q_ℓ that appear or do not appear in the corresponding set T_ℓ of the DB, as a function of block size b. The DB is the real database, where 470 elements where chosen to be the DB and the rest where chosen to be the queries. A "hit" means that a block of the query is one of the elements of T_ℓ.

b	# Blocks	Average # hits per query (STD)	Max # of misses per query
3	1157	1156.51 (0.66)	2
5	695	694.54 (0.60)	2
8	434	433.58 (0.55)	2
12	290	289.58 (0.49)	1

Frequency of number of block of the queries Q_ℓ that appear or do not appear in the corresponding set T_ℓ of the DB, as a function of the DB size. The DB is the real database, where random number of elements where chosen to be the DB and the rest where chosen to be the queries. Block size b is always 3, and so the number of blocks is always 1157.

DB size	#Queries	Average # hits per query (STD)	Max # of misses per query
276	224	1155.91 (0.62)	2
340	160	1155.90 (0.61)	3
400	100	1155.88 (0.62)	3
434	66	1155.95 (0.49)	2
470	30	1156.51 (0.66)	2

Table 4: Frequency of block miss as a function of block size and the database size.

[24] A. C. Yao. How to generate and exchange secrets (extended abstract). In *Symposium on Foundations of Computer Science, FOCS*, pages 162–167, 1986.

[25] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT*, pages 220–250, 2015.

A Block Misses

A.1 Frequency of Block Misses

Our approach ignores blocks of the clients that do not appear in the database. In Table 4, we report the frequency of block misses and show that it is a relatively rare event. The data below is on the real database, where we chose 30 sequences at random to be the queries, and the other 470 sequences to be the DB. We then built the tables T_ℓ, and run our protocol. Overall, more than 99.95% of the blocks of the queries do appear as one of the blocks in the DB. In fact, for more than half of the queries, all their blocks appear in the DB, and the maximal number of block misses per query that was observed is 3. In Table 4 we observe similar results even for small databases, and even when the number of queries and the number of sequences in the database are close.

A.2 On the Error Introduced by Block Miss

Even though that block misses are relatively rare events, it is still a question what to do in case they occurs. Assume that $Q_\ell \notin T_\ell$ for some block $\ell \in \{1, \ldots, n\}$. In the following, we compare between two possible approaches:

- The first approach is to compute the accurate distance between $\mathsf{ED}(Q_\ell, S_{i,\ell})$ for every $S_{i,1}, \ldots, S_{i,m}$. This introduces some additional complexity to the protocol, as we have to hide, both to the client and to the server, on which blocks Q_ℓ it holds that $Q_\ell \notin T_\ell$, as well as to compute edit-distances (of small blocks) in the online time. The resulting approximation function is as follows:

$$\mathsf{ApproxED}(Q, S_i) \;=\; \sum_{\ell=1}^{n} \Delta(Q_\ell, S_{i,\ell}), \text{ where } \Delta(Q_\ell, S_{i,\ell}) = \mathsf{ED}(Q_\ell, S_{i,\ell}) \tag{A.1}$$

- The second approach is the one we chose: we simply ignore these blocks. This results in the following approximation function (as we have already seen in Section 2.1):

$$\mathsf{ApproxED}(Q, S_i) \;=\; \sum_{\ell=1}^{n} \Delta(Q_\ell, S_{i,\ell}), \text{ where } \Delta(Q_\ell, S_{i,\ell}) = \begin{cases} \mathsf{ED}(Q_\ell, S_{i,\ell}) & \text{if } Q_\ell \in T_\ell \\ 0 & \text{otherwise .} \end{cases} \tag{A.2}$$

In Table 5, we show that the accuracy improvement is minor when choosing the first approach. This justifies our choice, as the overhead in computation for computing full edit-distances in case of block misses is significant. We report the average of 10 executions, where in each one of them we choose at random $\approx 4\%$ of the records as clients and the rest as the DB, which results in a database of size 480 on average. The reference genome is the global reference genome. The numbers are the percentages of correct queries, where a query is regarded as "correct" only if the exact correct k-closest sequences is returned, this is a very strict scoring, as even when the algorithm returns, for $k = 5$, a set with 4 correct ids and the fifth id has the same rank has the "correct" set, we count it as an incorrect answer.

	Approach I: Eq. (A.1)			Approach II: Eq. (A.2)		
Block Size b	$k=5$	$k=10$	$k=15$	$k=5$	$k=10$	$k=15$
3	99.78	96.99	94.37	99.78	96.99	94.37
5	100	97.63	94.59	99.78	97.63	94.55
8	100	98.01	94.80	99.68	97.90	94.95
12	100	97.69	94.80	99.78	97.53	94.91

Table 5: Percents of completely correct answers in the sets of closest k, for $k = 5, 10, 15$, comparing between Approach I: computing $\Delta(Q_\ell, S_{i,\ell}) = \mathsf{ED}(Q_\ell, S_{i,\ell})$ in case $Q_\ell \notin T_\ell$ (as in Eq. (A.1)), and Approach II: $\Delta(Q_\ell, S_{i,\ell}) = 0$ in case $Q_\ell \notin T_\ell$ (as in Eq. (A.2)).

DB	$\lvert Q_\ell \rvert$			Plurality I				Plurality II			Difference			
	Avg	STD	Max	Avg	STD	Min	Max	Avg	STD	Max	Avg	STD	Min	Max
Real	1.28	0.51	4	490.32	38.53	187	500	9.23	37.65	246	481.09	76.13	6	500
Sim	4.61	0.82	18	484.42	7.71	362	500	5.71	5.26	108	478.71	12.63	254	500

Table 6: Statistics on the sets Q_ℓ after applying BreakToBlocks on all sequences in the database, with $\mathsf{b} = 1$, in order to generate the synthesized reference genome. $|Q_\ell|$ is the number of different values in each block. Plurality I is the block value with the maximal number of occurrences. Plurality II is the block value with the second maximal number of occurrences. Difference are statistics about the difference between these two.

B On the Leakage of the Synthetic Reference Genome

In order to generate the synthetic reference genome, the server breaks all its sequences into block and choose for each block the plurality value the among the values in T_ℓ. This leaks information about the database. In this section we report some statistics about the plurality value in the database.

Our main goal in designing the algorithm for the generation of the synthetic reference genome is to maximize the accuracy of our approximation algorithm while at the same time, guarantee the privacy of individuals in the database. Our notion of privacy is inspired by differential-privacy [7,8]. Specifically, we consider an algorithm $\mathcal{A}$ that receives the database DB and produces the synthetic reference sequence (or, the breaking points for the reference genome, as in the hybrid case). We show that for the databases that we have in hand, DB, any "neighboring" database DB′ (a database that differs in only one record (sequence) from DB) it holds that $\mathcal{A}(\mathsf{DB}) = \mathcal{A}(\mathsf{DB}')$, namely, they produce the exact same reference genome, and therefore privacy of individual is preserved (but privacy of the population is leaked).[6]We also remark that the global reference genome was also synthesized from database of individuals.

We analyze the difference between the most frequent value for each block, and the second frequent value in each block. If the difference is large, it means that even if we change many individuals in the database, the most frequent value remains the same. In Table 6 we show statistics about the set of possible values Q_ℓ, on two databases of size 500, where the first is the real database, and the second are some 500 arbitrary records from the synthesized database.

The minimal difference in the real database is 6, which means that we can change up to 6 individuals in the database and the resulting reference genome would be completely the same (we recall that our algorithm performed remarkably well on the real dataset with the global reference genome, and therefore there is no reason to produce a synthesized one for it). The minimal difference in the synthesized database is large around 254, which means that there is no leakage on individuals even if we change and replace $\approx 20\%$ of the records. In Figure 1 we see the distribution of the difference between the most frequent value and the second most frequent value in a corresponding synthesized database.

[6]Nevertheless, we emphasize that the algorithm $\mathcal{A}$ does not satisfy the notion of differential-privacy, as we do not show this property for any two possible neighboring databases $\mathsf{DB}_1, \mathsf{DB}_2$ but rather only to some specific neighboring databases, namely, when either DB_1 or DB_2 are the databases we have in hand.

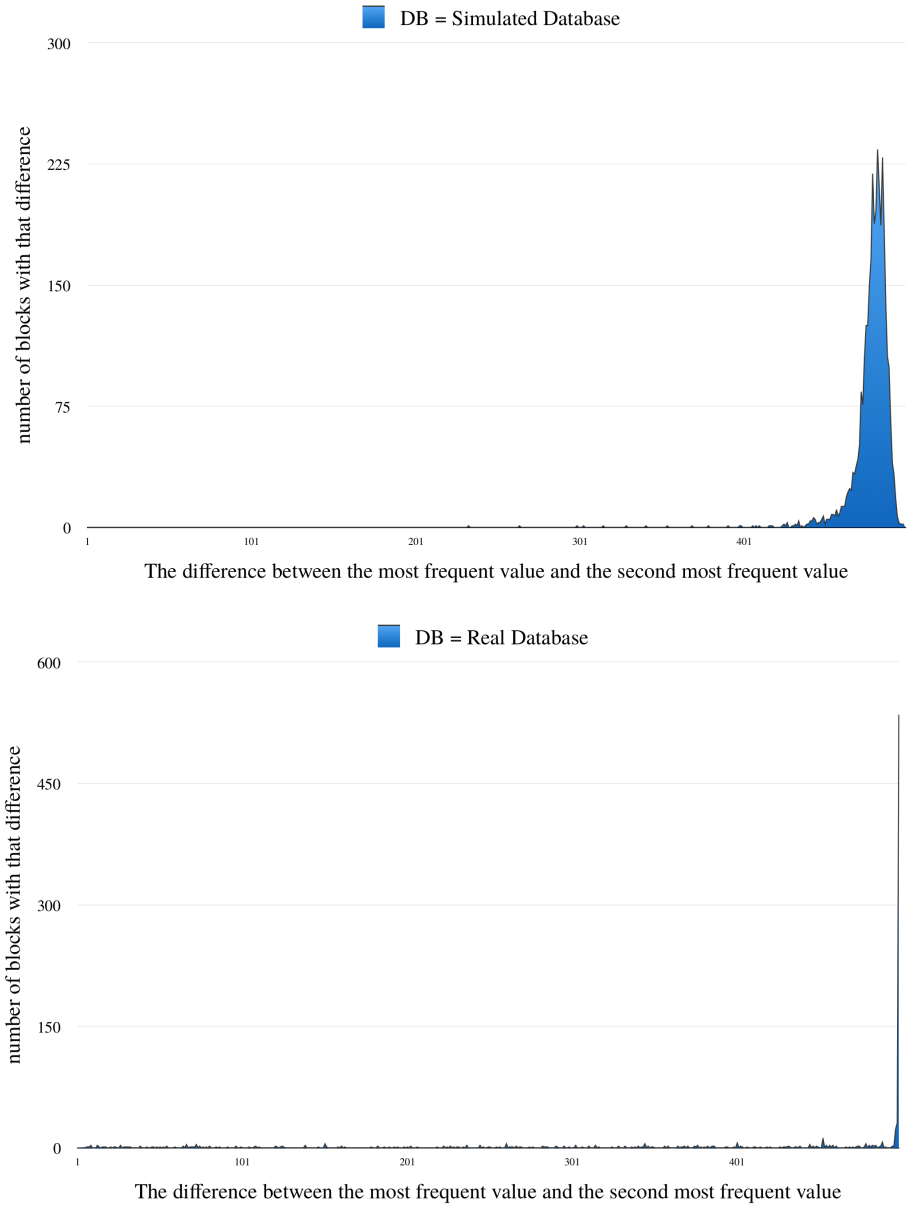

Figure 1: The difference between the number of occurrences of the most frequent value and the second most frequent value for a given block. The first graph is for 500 arbitrary records in the synthesized database, and the minimal difference observed is 233. The second graph is the real database, and the minimal difference observed is 6.

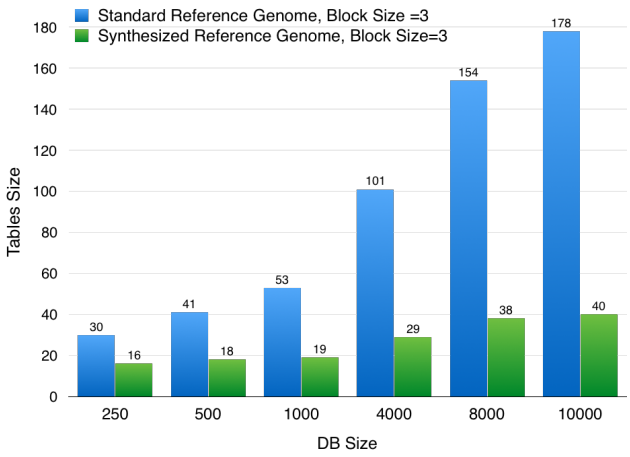

Figure 2: The maximum number of distinct values in any block for $\mathsf{b} = 3$ and varying database size.

C The Wagner-Fischer Algorithm

The WF algorithm [21] is based on dynamic programming, for computing the edit distance between two sequences $A = (\alpha_1, \ldots, \alpha_a)$ and $B = (\beta_1, \ldots, \beta_b)$.

The algorithm proceeds by preparing an $(a+1)$-by-$(b+1)$ matrix $D[\cdot, \cdot]$, where entry (i, j) is the edit-distance between the i-prefix of A and the j-prefix of B. The first row and column are initialized by $D[i, 0] = i$, $D[0, j] = j$ for all $0 \leq i \leq a$ and $0 \leq j \leq b$. Then for $1 \leq i \leq a$ and $1 \leq j \leq b$ the algorithm iteratively sets

$$D[i,j] = \begin{cases} \text{if } \alpha_i = \beta_j: & D[i-1, j-1] \quad \text{(match)} \\ \text{otherwise}: & \min \begin{cases} D[i-1, j-1] + 1 & \text{(substitution)} \\ D[i-1,\ j] + 1 & \text{(delete)} \\ D[i\ , j-1] + 1 & \text{(insert)} \end{cases} \end{cases}$$

and finally it returns the answer $D[a, b]$.

This procedure can be augmented to return not only the edit distance itself but also the sequence of operations that transforms A to B in $D[a, b]$ steps. Specifically, together with D we also prepare a matrix of pointers $PTR[\cdot, \cdot]$ (with the same dimension as D), that for each entry (i, j) points to the previous entry from which $D[i, j]$ received its value. Specifically, we initialize $PTR[0, 0] = \perp$, $PTR[i, 0] = (i-1, 0)$ for all $1 \leq i \leq a$ and $PTR[0, j] = (0, j-1)$ for all $1 \leq j \leq b$, and then for $1 \leq i \leq a$ and $1 \leq j \leq b$ we iteratively set

$$PTR[i,j] = \begin{cases} (i-1, j-1) & \text{if } D[i,j] \leq D[i-1, j-1] + 1 \\ (i-1,\ j) & \text{if } D[i,j] = D[i-1, j] + 1 \\ (i\ , j-1) & \text{if } D[i,j] = D[i, j-1] + 1 \end{cases}$$

where the first case corresponds to a match or substitution, the second corresponds to a delete, and the last case corresponds to an insert. When more than one condition applies, we break ties

toward the main diagonal. Namely, we prefer $(i\ , j-1)$ to the other options when $j > i$, prefer $(i-1,\ j)$ when $i > j$, and prefer $(i-1, j-1)$ when $i = j$.

The PTR table lets us trace on optimal path, starting from $PTR[a, b]$ and following the pointers to get both the alignment of the sequences A, B, as well as the corresponding operations (match, substitute, insert, delete).