# Anchoring Millions of Distinct Reads on the Human Genome within Seconds

Tien Huynh[†] Michail Vlachos[⋆] Isidore Rigoutsos[†]

[†] IBM T.J. Watson Research Center

[⋆] IBM Zürich Research Laboratory

## ABSTRACT

With the advent of next-generation DNA sequencing machines, there is an increasing need for the development of computational tools that can anchor accurately and expediently the millions of generated short DNA sequences (or reads) onto the genomes of target organisms. In this work, we describe ‘Q-Pick’, a new and efficient method for solving this problem. Q-Pick allows the rapid identification and anchoring of such reads with possible wildcards in large genomic databases, while guaranteeing completeness of results and efficiency of operation. Q-Pick requires very spartan memory and computational resources, and is trivially amenable to SIMD implementation; it can also be easily extended to handle longer reads, e.g. 75-mers or longer. Our experiments indicate that Q-Pick can anchor millions of distinct short reads against both strands of a mammalian genome in seconds, using a single-core computer processor.

## 1. INTRODUCTION

Next generation high-throughput DNA sequencing devices are causing a rapid transformation in the field of genome research by providing expedited and low-cost sequencing. While in the previous decade the efforts of the bioinformatics community targeted primarily the sequencing of the genetic material of organisms, with such tasks now completed, the focus has shifted to discovering commonalities and connections between newly sequenced molecules with respect to existing ‘reference’ genomes. For example, the mapping of the fragments from DNA sequences of cancerous cells on the human genome, can help isolate previously unknown cancer-initiating mutations.

Several genetic research companies, such as Illumina, Life Technologies, 454 Life Sciences and others, already offer new generation DNA sequencers. The sequencers when presented with a DNA (or RNA) sample, can produce millions of short DNA fragments from the given sample, also known as ‘reads’. The reads have short length, typically in the range between 20-40 nucleotides (nts); more recently, newer versions of the sequencing machines have begun generating longer reads (75 nts, or longer). With projected eventual outputs in the tens to hundreds of millions of reads from a given sample, the challenge now is the design of efficient tools that can quickly discover potential matches and anchor the reads on a reference genome.

The mapping of short sequence fragments on a genome has many **applications** including:

- Identification of the DNA sequence of a novel strain (natural or engineered), of a novel organism, or more importantly of a disease genome and matched normals [14, 25, 24].
- Transcriptome analysis of protein-coding and non-protein coding regions; this is known as RNA-seq [31, 3] and has been used extensively to determine which regions of a given organism’s genome give rise to transcripts.
- De novo assembly of an organism’s genome [6, 34].
- DNA methylation studies (following appropriate preprocessing of the sample) [5, 21, 13]

Additional applications where short-word indexing and search can be of great importance include design of siRNA’s (short interfering RNA’s) [7, 28, 33], junk DNA analysis [26], microarray probe design [9, 32, 19, 20, 29] and PCR primer design [8]. For a review of the various methods and applications, the reader is referred to [30] and references therein.

Traditional sequence mapping tools (such as BLAST or BLAT [11]) are unsuitable for mapping a massive amount of short-reads; since they were designed for matching longer sequences, and using default settings, they will miss (potentially many) short matches. Additionally, their running time is prohibitively long when presented with the large set of query sequences considered in the application at hand. Indicatively, a recent study [18] comparing various DNA matching techniques reports that for an experiment comparing approximately 10 million query sequences against a 5Mb human genome region, BLAST required more than 40 hours and BLAT approximately 6 hours, both matching 85% of the query sequences.

In comparison,newer approaches developed for aligning short-reads required from 2 to 8 minutes for the same experiments, and matched more than 90% of the given queries. Among the most prevalent approaches for short sequence matching are Eland [4], SOAP [18], fetchGWI [10], SHRiMP [27], Bowtie[12], Maq [15, 17], BWA [16] (see also [1, 2] for discussions and comparisons between various techniques). These approaches fall broadly into three categories: a) Methods that apply variants of the Smith-Waterman algorithm and hence are very slow for any practical purpose, due to the dynamic programming portion, b) approaches which perform exact Hamming matching through some index-based transformation, and c) hybrid approaches that trade accuracy for speed.

Our experiments, which are reported at the end of this paper, indicate that many of the short-word indexing approaches, may miss a number of potential matches in an effort to create a fast indexing scheme. The approach that we present in this work alleviates various shortcomings of previous efforts into a methodology that is both *fast* and *complete*. Our technique code-named `Q-Pick` (which stands for Quick-Pick), exhibits the following desirable characteristics:

1. it is external memory based and requires a minimal memory footprint. In our experiments, many competitive approaches surpassed the limits of the available memory for large data instances; however this was never observed for our methodology.

2. it does not require a large computer cluster or multi-core CPU to operate efficiently. Outstanding performance can be achieved even on a single-core/single-processor computer. Additionally, due to the program design its operations are easily amenable to massive parallelization.

3. it permits the rapid identification of matches and anchoring of corresponding reads, with millions of distinct short sequences being matched in seconds or minutes. Notably, Q-Pick is *flexible* in that it supports both exact and wildcard matches.

4. it guarantees completeness of the reported results, i.e. no candidate matches are missed; *all* locations where a read can be anchored, given the user specific settings, are reported. In the experimental section we demonstrate the extend to which competing approaches miss numerous *bona fide* hits, with some techniques missing as many as $50\%$ of the matching genomic locations.

In the upcoming sections we provide more technical details about the inner-workings of our solution. Finally, in the experimental section we include a thorough comparison of `Q-Pick` against many of the prevalent short-read mapping algorithms.

## 2. DESCRIPTION OF Q-PICK

### 2.1 Overview

`Q-Pick` is a new and highly optimized approach to performing very efficient matching of massive collections of potentially distinct sequence fragments (DNA or RNA) containing wildcards against very large genomic sequence databases. Based on our experiments and surveying the relevant bibliography, our method offers a uniquely competitive solution for matching short as well as longer reads. `Q-Pick` 's exceptional performance is achieved through the following technical contributions:

- **Data compression** through bit packetization. By exploiting the short DNA alphabet (4 letters), several sequence symbols can be compacted into a single computer word. Exploitation of 64-bit computer architectures can enhance even further the data packetization ratio.

- **Fast, bitwise comparison** between chunks of DNA sequences is possible by exploiting effectively the bit packetization. Multiple DNA symbols can be compared simultaneously using fast bitwise comparisons. The code is highly optimized to work with bit data representations, lending to the use of high throughput bitwise operators (left and right SHIFT, bitwise AND, OR, etc.).

- **Simplicity of implementation** through efficient joins of hash tables. Even though there exist advanced data structures, for search and storage, offering promising asymptotic bounds, e.g. suffix-trees, in practice the complexity of their implementation typically penalizes their performance thus limiting their real-world usefulness. Recently proposed suffix-trees approaches, such as Trellis [23], still still require large amounts of indexing space and cannot be used. `Q-Pick` is designed primarily with easiness of implementation in mind, allowing our contribution to be easily replicated.

- **Data pruning** is achieved through highly effective hashtable joins. Instead of sequentially scanning the whole genomic database for matches, the data are preprocessed and sequences sharing the same hash key (sequence prefix) are stored into the same hash bucket (clustered). A similar procedure is followed for the query patterns. Therefore, the search is essentially accomplished as a *hash join* that concentrates only on sequences that fall into hash buckets with the same key. This feature reduces significantly the runtime, by eliminating unnecessary comparisons, guiding very efficiently the search process. Finally, it is important to stress that the hashtable construction (an off-line step for the reference genome) is generally easily amenable to parallelization, if further performance improvement is deemed necessary.

Our dual hashtable index structure is partially inspired by the *spatial hash join* work which appeared in the database literature [22]. There, when joining two spatial datasets a spatial (R-tree) index is created on one of the datasets and the second one is only probed on the corresponding R-tree 'hyper-rectangles' that each point belongs to. Similarly, in our case, a static index is created off-line on the reference 'human' genome sequence (e.g. human, mouse, etc.) onto which the short sequence fragments will be anchored. A second index is created 'on-the-fly' given the aggregation of the short-reads. Finally, the equivalent hash buckets from the two indices (static and dynamic) are compared and the matching positions are reported. Additional optimizations that exploit the nature of the application, such as the short DNA alphabet, have been incorporated in our setting, leading to an exceptional search performance.

Figure 1 summarizes the various steps followed by `Q-Pick` . The left side illustrates the process for creating the static genome index and the right side depicts the creation of the dynamic index based on the different set of DNA fragments returned from the sequencing process. In the following sections we will analyze in more detail the creation of both indices, the binary encoding of sequences on the index and the search process.

### 2.2 Data Representation and Bit Packetization

Our DNA search/matching process involves two sources of genomic sequences:

- A *target, or reference, database* which contains the complete genomic sequence (complete, or partially assembled contigs) of the organism of interest, onto which we would like to perform search.

- A *query database* which consists of a possibly very large collection of short reads that are assumed for generality to be of variable length and to contain wildcards at specific positions; these sequences or reads would be typically generated by any of the high-throughput sequencers (e.g. 454/Roche's FLX system, ABI's Solid system, Illumina's Genome Analyzer, etc.). Those positions within the read where the quality of the sequencing, as estimated by the provider's quality analysis software, has fallen below a predetermined threshold are replaced by a wildcard prior to the mapping/anchoring step. Our task is to discover in a fast, exhaustive and efficient manner the instances of the generated short sequences on the genomic target database.

The **"reference" or "target" database** will generally consist of multiple chromosomes, each being one long strand of DNA. Note that

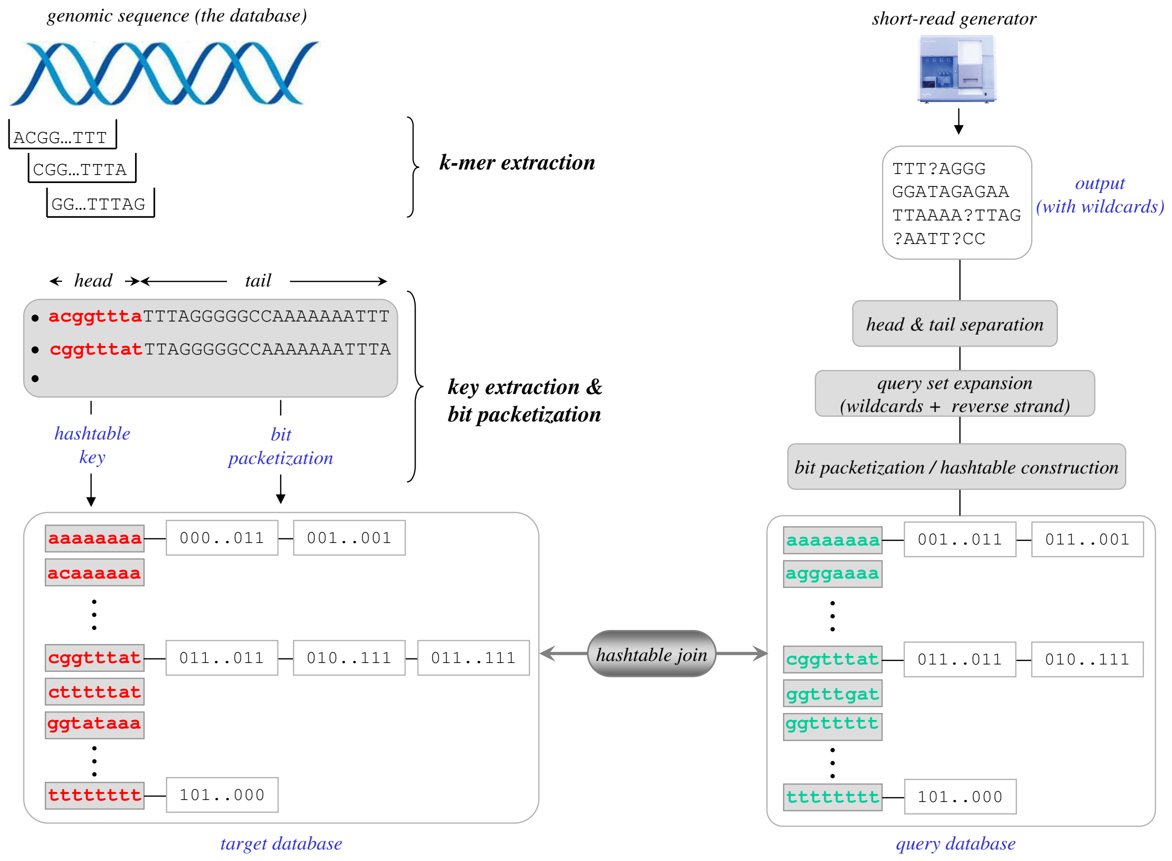


**Figure 1.** Overview of the Q-Pick approach

we are interested in anchoring the generated reads on both the forward and reverse strands of each chromosome: instead of encoding separately the two strands of each chromosome, we opt to reduce the indexing-space requirements by appropriately processing the query database (only single strand) and incorporating additional bookkeeping (this is explained in detail below).

The **"query" database** itself is generated from the multiple sequences to be anchored. Next, and due to the different sequence extraction and encoding schemes that have to be deployed in each case, we provide separate descriptions on how to represent and encode sequences from these two databases,

### 2.2.1 *Encoding the "target" database*

We begin by describing the pre-processing, data representation and encoding of the reference genome into a binary format. This will allow its compact representation and also effectively determine the mapping functions of the constructed hashtable index.

Let's assume for the moment that $L_{read}$ represents the length of a read and that the maximum length of all reads to be matched and anchored on the genome is $L_{max}$. Given $L_{max}$, all N-grams with this size are extracted from the target database (e.g. human genome). This can be achieved simply by positioning a sliding window of length $L_{max}$, extracting the appropriate subsequence, sliding the window one position to the right and repeating this step. It is important to stress that all possible subsequences of length $L_{max}$ are extracted, and thus one can guarantee that no potential matching subsequences will be missed. The process is repeated in turn for each of the different chromosomes in the genome of interest (e.g. 24 in the case of the human genome). Recall, that only the forward strand of each chromosome is processed during this encoding stage.

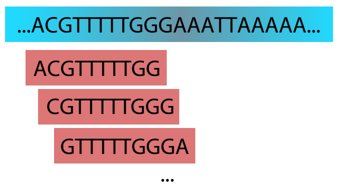


**Figure 2.** N-gram extraction from the reference genome

Subsequently, each of the extracted N-grams $g$ is broken into two logical parts: the *head* and the *tail* with lengths $L_h$ and $L_t$, respectively. The head will serve as the indexing key into the formed hashtable (i.e. it will "point" to the appropriate bucket) whereas the tail corresponds to the remaining nucleotides of the N-gram, and represents the content that will be stored in the hashtable bucket pointed to by the head. In our implementation, we use $L_h = 14$ nucleotides as the key; given that the DNA comprises four bases (A,C,G,T), the hashtable can have a maximum of $4^{14} = 268,435,456$ buckets. An additional binary table, called "Marker Table", with $4^{14}$ positions will also be employed, indicating the empty and non-empty bins of the hashtable. This table will be used later on when performing search of the hashtable.

Each nucleotide $g(i)$ of the head will be mapped into two binary digits using a function $\phi$ as follows:

$$\phi(g(i)) = \begin{cases} 00 & \text{if } g(i) = A \\ 01 & \text{if } g(i) = C \\ 10 & \text{if } g(i) = G \\ 11 & \text{if } g(i) = T \end{cases}$$

Therefore, for an extracted N-gram $g$, if $\bigcup$ is a string concatenation function, then the mapping function $f$ to the appropriate hashtable position is defined as:

$$f(g) = \phi(g(1)) \bigcup \phi(g(2)) \bigcup \ldots \bigcup \phi(g(L_k))$$

For example, as shown in Figure 3 for the pattern:

```
ACGTACGTACGTACGTACGACACGTAC
```

its head would be:

```
ACGTACGTACGTAC
```

and will be encoded as 0x01b1b1b1 in hexadecimal, which corresponds to bucket number $28,422,577$ in the hashtable and indicates that the tail of the extracted N-gram, i.e. `GTACGACACGTAC`, will be stored in that bucket of the hashtable. Clearly, $L_t$ satisfies the following condition: $L_h + L_t \leq L_{max}$.

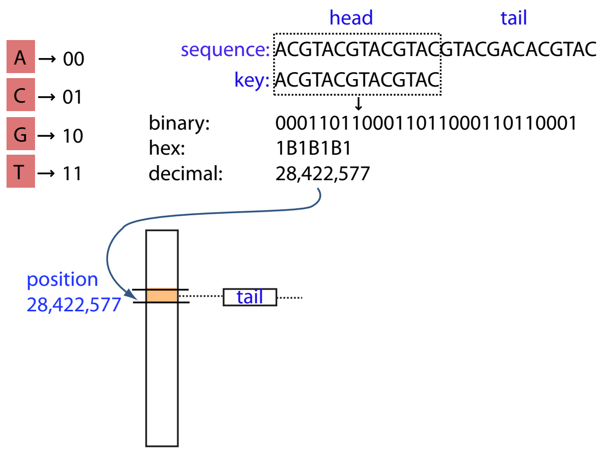


**Figure 3.** Encoding and binarization of the head of the extracted N-grams (hashtable key)

We point out that the distinction between head and tail of the N-gram serves a dual purpose:

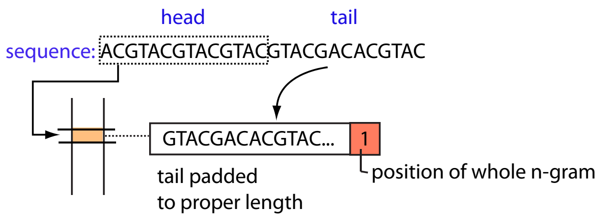
1. It serves as a way of extracting a key for the hashtable, and
2. It results in a reduction of the dataset size, since the head does not need to be explicitly stored. Therefore, for all sequences that share the same head, only their corresponding tail portions need to be recorded in the corresponding bucket that is pointed to by the head portion of the N-gram.

For the tail, which is the portion that will actually be stored in each bucket, we employ a different encoding scheme in order to make it amenable to fast bitwise operations. Now, we replace each symbol of the tail by a 4-bit vector: the i-th bit (i=1,2,3,4) will be 1 if the symbol at hand corresponds to the i-th letter of the 4-letter alphabet, and 0 otherwise. We note also that if the tail of the extracted N-gram does not fill the fixed maximum tail length (i.e. if $L_t < L_{max} - L_h$) then the remaining positions are padded with wildcards, which will allow them to be matched with any nucleotide. For the scope of this paper, wildcards are denoted with *dots*; wildcards need their own binary representation *dots*. All these considerations, result in the following bit-vector encoding function $\psi$ of the tail nucleotides:

$$\psi(g(i)) = \begin{cases} 0001 & \text{if } g(i) = A \\ 0010 & \text{if } g(i) = C \\ 0100 & \text{if } g(i) = G \\ 1000 & \text{if } g(i) = T \\ 0000 & \text{if } g(i) = .\ \text{(wildcard)} \end{cases}$$

In our experiments we assumed a maximum length $L_{max} = 30$ the query reads, and thus for the N-grams that are extracted from the target database. Consequently, the length of the tail $L_t = L_{max} - L_h = 30 - 14 = 16$ nucleotides which can be easily packed into a 64-bit long integer. Therefore, later when comparing query sequences with the already indexed ones, we can deploy fast bitwise operations of 64-bit architectures [1].

It is important to point out Q-Pick can be easily modified to handle the kinds of reads produced by the newer platforms, whose average length exceeds the value of $L_{max} = 30$ that we have assumed for this implementation. To handle this case, straightforward modifications will be required during the encoding of both the target and query databases:in order to avoid interrupting the flow of the presentation, we discuss these modifications in the end.



head
tail
sequence: ACGTACGTACGTACGTACGACACGTAC
GTACGACACGTAC...
1
tail padded to proper length
position of whole n-gram

**Figure 4.** Contents of the hashtable buckets (tail of the N-gram)

*Example:* Suppose that the target database consists of the sequence:

```
ATAGACTAAAAAAAAAAAAAAATT
```

Since $L_{max} = 30$, all 30-grams that contain at least $L_h + 1 = 15$ characters and are no longer than 30 characters are extracted. As we discussed, N-grams that are shorter than than $L_{max}$ characters will be padded with dots (i.e. wildcards) up to the length $L_{max}$. In the below, we will denote the head (resp. tail) part of the extracted subsequences using lower (resp. upper) case letters. For simplicity of presentation we do not depict the binary representation of the sequence but only its symbolic one. Finally, on the record containing the tail part of the N-gram, the shown trailing number pinpoints the location of that N-gram in the given sequence. For the example sequence shown above, the N-grams that will be extracted are:

```
atagactaaaaaaaAAAAAAAATT...... 1
tagactaaaaaaaaAAAAAAATT....... 2
agactaaaaaaaaaAAAAAATT........ 3
gactaaaaaaaaaaAAAAATT......... 4
actaaaaaaaaaaaAAAATT.......... 5
ctaaaaaaaaaaaaAAATT........... 6
taaaaaaaaaaaaaAATT............ 7
aaaaaaaaaaaaaaATT............. 8
aaaaaaaaaaaaaaTT.............. 9
aaaaaaaaaaaaatT............... 10
```

[1]Obviously, for longer extracted N-grams, the process remains identical with the exception that fewer sequences can be compared at once.

All the N-grams that share the same key (i.e. the first $L_h = 14$ characters) will be clustered into the same hashtable bucket. The resulting hashtable entries will contain the trail subsequence and also its original position on the chromosome. The hashtable would look like (the string within the square brackets correspond to the index of the hashtable bucket):

```
[aaaaaaaaaaaaaa]<ATT............., 8>, <TT.............., 9>
[aaaaaaaaaaaaat]<T..............., 10>
[actaaaaaaaaaaa]<AAAATT.........., 5>
[agactaaaaaaaaa]<AAAAAATT........, 3>
[atagactaaaaaaa]<AAAAAAAATT......, 1>
[ctaaaaaaaaaaaa]<AAATT..........., 6>
[gactaaaaaaaaaa]<AAAAATT........., 4>
[taaaaaaaaaaaaa]<AATT............, 7>
[tagactaaaaaaaa]<AAAAAAATT......., 2>
```

```
for each chromosome C of the genome
{
  for position=1:length(chromosome C)
  {
    // extract N-gram of length L
    // at current position
    g = extract(C, position, L);
    H = Head(g); // extract head
    T = Tail(g); // extract tail

    key = binarizeHead(H); // head encoding
    content = binarizeTail(T); // tail encoding

    // place in proper hash bucket
    pointer = Hash_TargetDB<key>;
    append(pointer) -> {c: content, p:position}

    // indicate that bucket has content
    Marker_Table<key> = 1;
  }

  // record each bucket to a separate
  // file on disk
  for each non_empty(Marker_Table)
    write_to_disk(Hash_TargetDB<key>);
}
```

**Figure 5.** Algorithm for creation of the database on the reference genome

An overview of the algorithm for the creation of the reference database is given in Figure 5. It is important to stress here that only non-empty hashtable buckets need to be recorded and stored on the disk. Moreover, the process of transcribing the hash buckets on the disk can be done in an efficient and memory-friendly way by exploiting the performance benefits of *batch writes* on the disk. A memory buffer is kept for each hashtable bucket and the corresponding portion of the bucket is directed to the disk not constantly, but only when the memory buffer size of the bucket is filled. The efficiency of such a construct has also been noticed in the construction of *seeded trees* when performing external memory joins [22].

Next, we describe how the set query sequences are encoded and stored on a different query hashtable.

### 2.2.2 *Encoding the query database*

The query database will typically comprise millions of distinct sequences, which will be generated by the sequencing platform. We now formally describe the creation of the query sequences' hashtable.

Similarly to the creation of the target database, each query sequence is logically divided into a head and a tail part, using two (2) and four (4) bits for the encoding, respectively, identically to what was done for the reference genome. However, there are two differences between the target database and the query database:

- the typical query sequences may contain one or more wildcards ("don't care") symbols. This is unlike the case of the N-grams that we extracted from the target database and which contain wildcards only at the terminating positions where they serve as padding elements.
- query sequences can have variable length, compared to the fixed length N-grams that are extracted from the genomic database.

Wildcards are treated differently depending on whether they exist in the head or the tail of a sequence in the query database; this is because the head is used as the key of the hashtable mapping function.

- **Case 1**: One or more wildcards exist in any of the $L_h$ positions of the head of the query. In this case each wildcard is *expanded* into the relevant A,C,G,T symbols, in order to form all appropriate hashtable keys. For example, if the head of query $q$ is `.TTTTTTTTTTTT.` then the expansion will generate a total of $4 \times 4 = 16$ potential head sequences and the corresponding hashtable bucket indices where the tails of all the patterns that share the same head `.TTTTTTTTTTTT.` are to be found. Therefore, the same tail of query $q$ will be hashed onto a total of 16 positions of the query hashtable index. An illustration of the expanded sequences is shown in Figure 6
- **Case 2**: One or more wildcards exist in any of the $L_t$ positions of the tail of a query. In this case, each such wildcard will be encoded using the bit vector 0000 (recall that 4 bits are utilized to encode each of the nucleotides and the wildcard). The reason for the encoding will be become apparent shortly, when we explain the matching function utilized to join the query and reference database hashtables. This encoding for the wildcard, as will be shown later on, can exploit bitwise operations for fast sequence comparison.

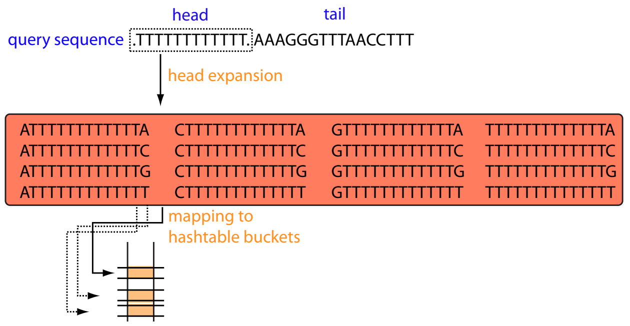


**Figure 6.** Query Database: Expansion of the wildcards on the head of a query sequence.

Finally, since query patterns may be of variable length, and similarly to the trailing N-grams of the target genome, query tails that are shorter than the $L_{max} - L_h = 30 - 14 = 16$, will be padded with wildcards (i.e. zeros).

**Example:** The 16-letter tail the 30-nucleotide-long query

```
ACGTACGTACGTACGTACG.AC..ACGTAC
```

is `GTACG.AC..ACGTAC` and will be encoded as the hexadecimal number `0x4812401200124812`. If the tail of the pattern were `GTACG.AC..AC`, which is only 12-nucleotides long, it would have been padded-up with four '.' at the end, and encoded as `0x4812401200120000`.

### 2.2.3 *Indexing of the Query Database for handling Forward and Reverse strands of Target Genome*

Recall that the target DNA consists of two strands; the forward and the reverse, both of which need to be searched for instances of a query sequence. In lieu of explicitly encoding the reverse strand in the target database hashtable, which would effectively double the size of indexed data, we chose to simply expand the query set by generating and encoding the reverse complement of each query. This leads to a reduction of overall storage requirements because the query database will typically be smaller in size. Therefore, instead of searching the forward and reverse strands of a given chromosome for instances of a query, we search only the forward strand for instances of a query or of a queryŠs reverse complement. This design strategy results in a much more compact representation, and, by extension, contributes to the realized speed gains.

This mapping is performed as follows; if $\hat{s}$ denotes the reverse complement of symbol $s$, with $s \in \{A, C, G, T\}$, then the reverse complementary sequence of an N-gram $g$ can be written as:

$$\hat{g} = \hat{g}_n \cup \hat{g}_{n-1} \cup \ldots \cup \hat{g}_1, \text{where } g = g_1 \cup g_2 \cup \ldots \cup g_n$$

The general concept is shown in figure 7.

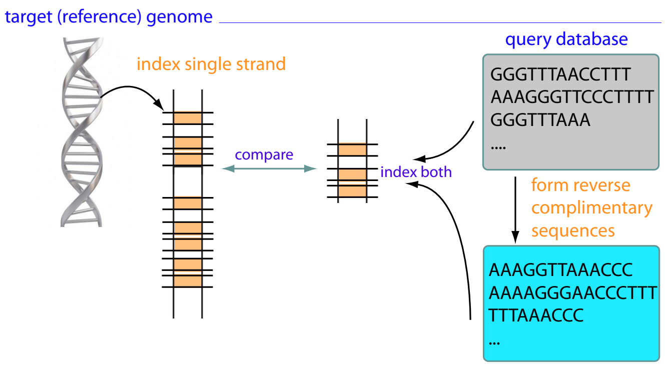


**Figure 7.** Accommodating the reverse strand of the DNA through expansion of the (smaller) query set.

**Example:** Suppose that we are presented with the following 4 query sequences:

```
1: ACAAAAAAAAAAAATT
2: AC.AAAAAAAAAAAA
3: TTTTTTTTTAGTCTAG
4: TTTTTTTTTAGTCTAT
```

This initial set of 4 queries will be expanded into the following 8 sequences:

```
1: acaaaaaaaaaaaaTT
2: aattttttttttttGT
3: ac.aaaaaaaaaaAA
4: tttttttttttt.GT
5: ttttttttagtctAGG
6: cctagactaaaaaaAAA
7: ttttttttagtctAT
8: atagactaaaaaaaAA
```

For example, the first sequence (ACAAAAAAAAAAAATT) will generate also the reverse complimentary sequence AATTTTTTTTTTTTGT. In each case, the lower case letters indicate the portion of the query that corresponds to the head and will be used as the key. The tail is shown using capital letters. As described above, each of the wildcards contained in the head portion will be dereferenced into each of the 4 possible DNA symbols in turn. In the above example, we have 2 such query sequences (♯3 and ♯4) each containing a single wildcard; thus, query ♯3 will be replaced by 4 new strings as will query ♯4, resulting in a grand total of $8 - 2 + 4 + 4 = 14$ strings.

The complete process for encoding the query sequences is captured in Figure 8.

```
for each sequence S
{
  for position=1:length(S)
  {
    // generate reverse complement sequences
    {candidate sequences} = reverseComplement(S);

    for each candidate sequence s
    {
      H = Head(s); // extract head
      T = Tail(s); // extract tail

      {keys} = expand_wildcards(H);
      {keys} = binarizeHead({keys});
      content = binarizeTail(T);

      for each key
      {
        // place in proper hash bucket
        pointer = Hash_QueryDB<key>;
        append(pointer) -> {c: content};
      }
  }
}
```

**Figure 8.** Algorithm for creation of the query database

## 2.3 Search Process

After the hashtables of each chromosome of the target database and the hashtable of the query set have been created, the search process is initiated as a hashtable join of the buckets with the same key.

First off, the “Marker" table of the target hashtable which indicates the non-empty buckets in the reference database is loaded in memory. The process proceeds by scanning through the buckets of the query hashtable and examining the corresponding position on the Marker Table (see Figure 9). If the Marker table indicates that there are entries in the corresponding bucket on the genome hashtable, then that bucket is retrieved and its elements (tails of the extracted N-grams) are compared with the bucket of the same key from the query hashtable (tails of the query sequences).

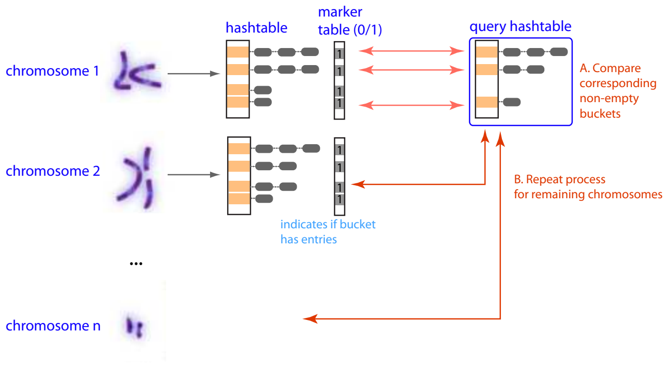

hashtable
marker table (0/1)
query hashtable
chromosome 1
chromosome 2
...
chromosome n
A. Compare corresponding non-empty buckets
B. Repeat process for remaining chromosomes
indicates if bucket has entries

**Figure 9.** Search process through hash-join

The comparison between the sequences contained in the two buckets can be achieved in a very fast manner utilizing bitwise operations. If we denote the tail of one of the many target genome N-grams by *target_tail* and the tail portion of a query sequence by *query_tail*, then a match between the two is successful iff:

$$(< query\_tail > \& < target\_tail >) == < query\_tail >$$

where $\&$ is the bitwise AND operator. We recall from Section 2.2.2 that wildcards are encoded as as 0000. Therefore, the above matching function will hold true also in the presence of wildcard symbols. Moreover, thanks to the right padding with wildcards, the two operands are guaranteed to have the same length so the operation will always be well-formed.

Finally, the the comparison between same-key buckets of the query and target hashtables is repeated for the hashtables of each chromosome in turn. The whole search algorithm is given graphically on Figure 9 and pseudocode is provided in Figure 10.

```
load the MARKER array of the target hashtable
for each bucket B1 in the query hashtable
{
  fetch the query bucket <key> from B1
  if (MARKER[<key>] == 1)
  {
    load B2 bucket with same <key> as B1
    for each <query_tail> in B1
    {
      for each <tail> in B2
        if (<query_tail> & <tail>) == <query_tail>
            print <match, location>
    }
  }
}
```

**Figure 10.** Join of query and target database hashtables

The search process through the join of the two hashtables is very effective, because the majority of the buckets in the target database hashtable contain only a handful of entries. Figure 11 shows on a log-log plot a histogram of the bucket sizes for the first human chromosome. We note that $98.5\%$ of the buckets contain less than 10 entries, indicating that our choice of key and key length, properly separates the different subsequences. Similar results are observed for the remaining chromosomes, as well.

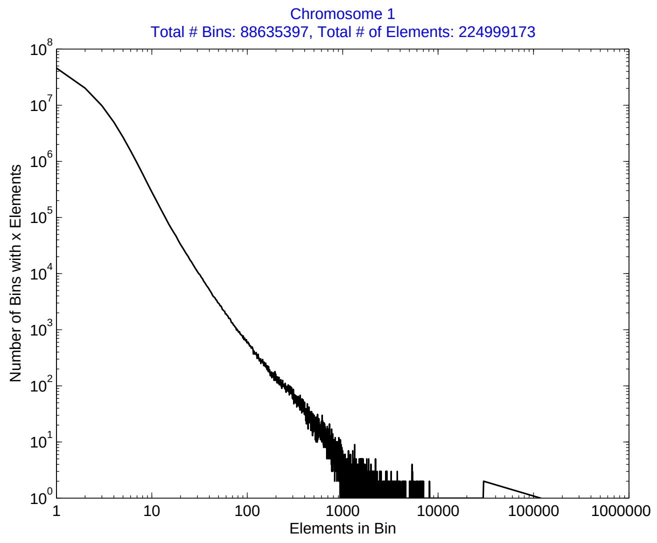


**Figure 11.** Log-log plot of histogram indicating the sizes of the different hash bins. We observe that the majority of the bins contain very few elements (typically 1), which suggests that only few comparisons need to be made between the corresponding database and query bins.

## 2.4 Visualizing the results

After the search process is complete, the matching positions and the corresponding sequences are stored on the disk. Given that the matching positions are listed using a chromosome identifier and global coordinates within the chromosome, the matches can be trivially rewritten using, e.g. the BED format (see `http://genome.ucsc.edu/FAQ/FAQformat`) and uploaded on the UCSC Genome Browser.

We have also instrumented an interactive visualization interface to facilitate a somewhat different kind of exploration of the matching positions, through drill-up/drill-down capabilities, as shown in Figure 12. A histogram of the matching positions with respect to the reference genome length is displayed at the initial screen, while the matches of the queries along with the respective positions are indicated at the bottom of the window. The user can select the bin of interest (range of positions in the genome) and drill down in order to 'zoom in' on a specific range of query matches. Additionally, the user has the ability to adjust the width of the histogram bins, effectively changing the resolution of the view.

The histogram based view of the matches is efficient and effective. At a later incarnation of our GUI, we plan to incorporate annotations corresponding to the regions of the genome where a read has been anchored, a functionality that will certainly strengthen and augment any subsequent analysis.

## 2.5 Extending Q-Pick to handle long reads

There are several, rather straightforward ways in which we can extend Q-Pick so that it can handle longer reads. One such way is described here. Let us assume that the query set now comprises variable-length reads whose maximum length $L_{max}$ is greater than 30, e.g. $L_{max} = 75$. The following simple modification will suffice for the purpose and is dictated by our goal of keeping `Q-Pick`'s memory and storage requirements to a minimum. If we were to continue storing the tail of an N-gram in each bucket of the target genome's hashtable, memory (and storage) requirements would increase proportionally with $N$; for large values of $L_{max}$ the storage requirements would be substantial. Instead, in the hashtable bucket that is pointed to by the N-gram's head, we enter only the N-gram's position in the genome (but not the sequence of the N-gram's tail). The encoding of the query database remains the same as before with each entry requiring exactly $4 \times (L_{max} - L_h)$ bits. During the search stage, and prior to the comparison step, the tails of all entries of a given hashtable bucket for the target genome will need to be retrieved through an additional dereferencing step: this is a fast operation and incurs a negligible cost compared to the great savings in storage and memory requirements and the ability to handle much longer reads in the query set.

## 3. EXPERIMENTS

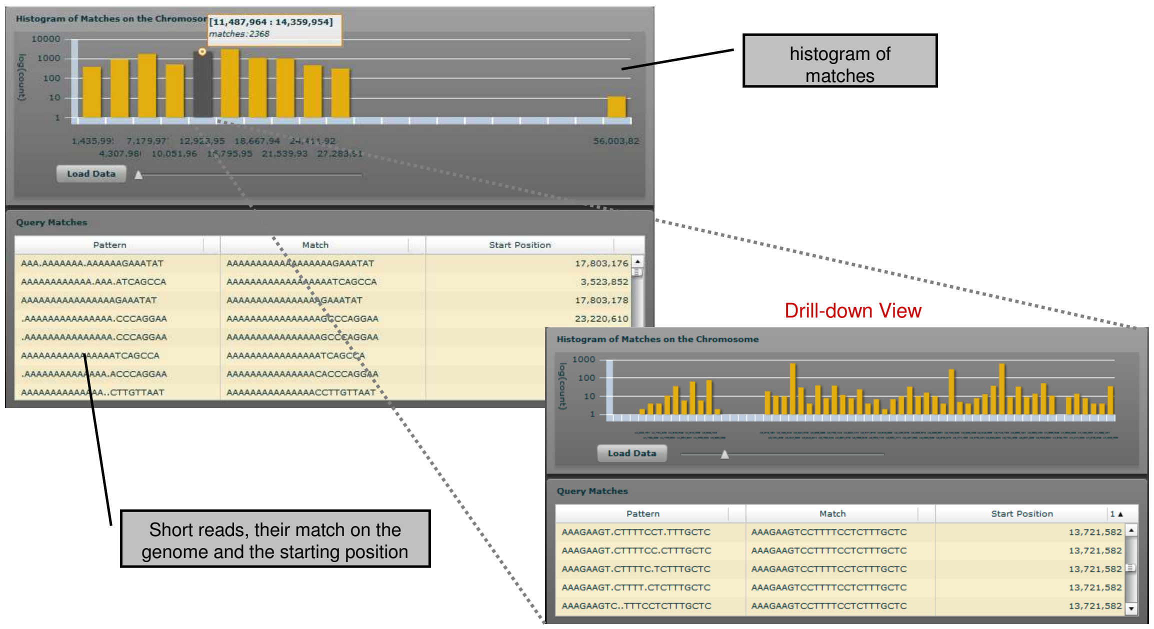
In this section, we elaborate on the performance capabilities of `Q-Pick` through a battery of carefully designed experiments. We conduct extensive scale-up experiments with millions of *distinct* query sequences. As our target databases we utilize the Mouse (`NCBIM37.49`[2]) and the Human (`NCBI36.42`[3]) genomes.

## 3.1 Comparison with other techniques

We provide comparisons with other prevalent short-word matching techniques that are currently available. The gamut of methods that we compared with `Q-Pick` was limited by our ability to port them to a SuSE Linux Enterprise Server: we only used those applications that could be ported with only moderate effort and could be run

[2]`ftp://ftp.ensembl.org/pub/release-49/fasta/mus_musculus/dna/`

[3]`ftp://ftp.ensembl.org/pub/release-42/homo_sapiens_42_36d/data/fasta/dna/`



Query Matches

| Pattern | Match | Start Position |
|---|---|---|
| AAA.AAAAAAA.AAAAAAGAAATAT | AAAAAAAAAAAAAAAAAAGAAATAT | 17,803,176 |
| AAAAAAAAAAAA.AAA.ATCAGCCA | AAAAAAAAAAAAAAAAAATCAGCCA | 3,523,852 |
| AAAAAAAAAAAAAAAAGAAATAT | AAAAAAAAAAAAAAAAGAAATAT | 17,803,178 |
| .AAAAAAAAAAAAAAA.CCCAGGAA | AAAAAAAAAAAAAAAAGCCCAGGAA | 23,220,610 |
| .AAAAAAAAAAAAAAA.CCCAGGAA | AAAAAAAAAAAAAAAAGCCCAGGAA | |
| AAAAAAAAAAAAAAAATCAGCCA | AAAAAAAAAAAAAAAATCAGCCA | |
| .AAAAAAAAAAAAAA.ACCCAGGAA | AAAAAAAAAAAAAAACACCCAGGAA | |
| AAAAAAAAAAAAAA..CTTGTTAAT | AAAAAAAAAAAAAAACCTTGTTAAT | |

Query Matches

| Pattern | Match | Start Position |
|---|---|---|
| AAAGAAGT.CTTTTCCT.TTTGCTC | AAAGAAGTCCTTTTCCTCTTTGCTC | 13,721,582 |
| AAAGAAGT.CTTTTCC.CTTTGCTC | AAAGAAGTCCTTTTCCTCTTTGCTC | 13,721,582 |
| AAAGAAGT.CTTTTC.TCTTTGCTC | AAAGAAGTCCTTTTCCTCTTTGCTC | 13,721,582 |
| AAAGAAGT.CTTTT.CTCTTTGCTC | AAAGAAGTCCTTTTCCTCTTTGCTC | 13,721,582 |
| AAAGAAGTC..TTTCCTCTTTGCTC | AAAGAAGTCCTTTTCCTCTTTGCTC | 13,721,582 |

**Figure 12.** Visualization interface for `Q-Pick`

| | Q-Pick | fetchGWI | SOAP | Eland | BWA | Bowtie |
|---|---|---|---|---|---|---|
| *Hits* | $10,611,858$ | $10,611,856$ | $10,573,528$ | $5,597,799$ | $10,611,858$ | $10,611,858$ |
| *Misses* | $0$ | $2$ | $38,330$ | $5,014,059$ | $0$ | $0$ |
| *Time* | 79 sec | 186 sec | 4728 sec | 555 sec | 149 sec | 331 sec |

**Table 1.** Comparison between various short-word matching approaches

to completion without crashing. The CPU powering the server was a 64bit PowerPC, running at 1.4Ghz; clearly, running Q-Pick on machines with faster CPUs will incur further improvements over what we report below.

The list of similar tools that we used for our comparisons includes:

- **Eland** [4] is a short-read mapping tool that is provided by Illumina as accompaniment to their sequencing platforms.

- **SOAP** (Short Oligonucleotide Analysis Package) [18], is an open-source tool, that has attracted a lot of attention in the bioinformatics community due to its expedient runtime and high specificity of matches.

- **FetchGWI** [10] was specifically designed for short sequence mapping within genomes in a rapid manner. It is also one of the few approaches that makes claims about completeness of matching (i.e. that all matching positions will be found and reported).

- Finally, **Bowtie** [12] and **BWA** [16] are slightly different than the other techniques in that they derive their speed by working with a genome that has been represented using the Burrows-Wheeler transform.

The results of our comparison are shown in Table 1. For this particular experiment the query database comprised 1 million *distinct* 22-mers, that were searched against both strands of the mouse chromosome 1 (a total of 197M x 2 nucleotides).
From this first experiment we can draw several conclusions. First, `Q-Pick` can be *between* $2-60$ *times faster* than existing approaches, at the same guaranteeing the exhaustive reporting of all anchoring points for the given query set. `Q-Pick` , `Bowtie` and `BWA` were the only approaches that reported all matches: the remaining methodologies missed numerous *bona fide* anchroring positions for the employed query set.

It is important to stress here that the reported times represent the end-to-end time for performing all the necessary preprocessing of the query set, searching anchoring, and finally the reporting of the results. While, `Q-Pick` and `fetchGWI`, `Bowtie` and `BWA` report all hit locations, `SOAP` and `Eland` report counts for the hits of a given read but the location of the 1st hit only. In fact, the actual search time of `Q-Pick` is a mere 19 seconds: the remaining 60 seconds are consumed by the writing to the disk of the ~10 Million matching positions and corresponding genomic sequences to disk.

There is a particularly important point to be made here that relates to comparing performance measures. Indeed, “reporting" a given number of hits requires that they be written to the output device and the time required to do this is proportional to the size of the generated output. Consequently, a method that simply reports counts of hits instead of the actual hits themselves is at a performance advantage compared to a method such as `Q-Pick` that reports all hits in the form of from/to coordinates and the matching genome string. What this means in practical terms is, that the true speedup that `Q-Pick` achieves when compared to `SOAP` or `Eland` is substantially better than what is indicated by the entries of Table 1.

In the following section we provide a more thorough comparison with `fetchGWI`: of the available techniques, the latter is the closest methodologically to `Q-Pick` (use of indexing, hash-joins, etc.).

## 3.2 Comparison with fetchGWI

First, we examine the scale-up performance of the algorithms under increasing query set cardinalities. Figure 13 reports the timing experiment when posing query sets of variable cardinalities against human chromosome 1. The query set sizes range from 1 million to 10 million short sequences. `Q-Pick` outperforms `fetchGWI` by an order of magnitude in runtime. For example, for the 10 million query set `Q-Pick` requires just 86 seconds, while `fetchGWI` executes for more than 23 minutes, on the same platform.

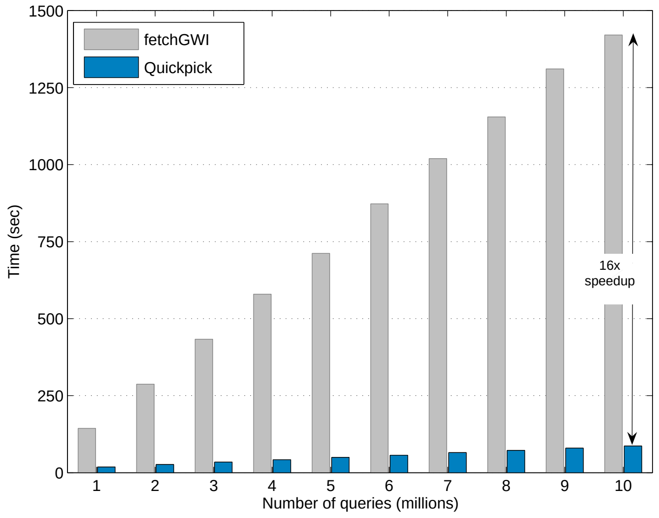


**Figure 13.** Runtime comparison between `Q-Pick` and `fetchGWI`. `Q-Pick` provides an order of magnitude improvement in response time. `Q-Pick` returns the matching positions of 10 million queries in just 86 seconds compared to 23 minutes required by `fetchGWI`.

An interesting discovery of this first experiment surfaces when we compare the cardinality of matching positions for the query sets. Figure 14 shows the number of discovered results for the two approaches. Note that `fetchGWI` misses approximately $2\% - 3\%$ of the matching positions. This result is counter to the completeness claim of the original `fetchGWI` publication. However, we suspect that this is very likely the result of an implementation bug in the current release of `fetchGWI`.

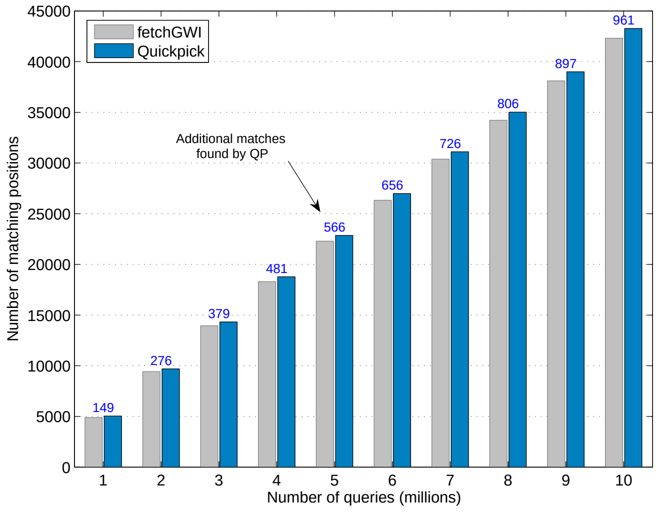


**Figure 14.** Number of matches found by `Q-Pick` and `fetchGWI`. The current implementation of `fetchGWI` fails to find all the matching positions of the query set; on the tested queries it missed approximately $2 - 3\%$ of the answer set.

The following experiments elaborate and examine performance issues of our framework against different parameters of the search problem.

## 3.3 Varying the number of wildcard positions

`Q-Pick` supports both exact word match and matches with wildcard positions. As the number of wildcard positions increases, so does the search time, since there is an additional increase in the hashtable generation time for a given query set; recall that each wildcard (dot) in the key has to be expanded to the potential A,C,G and T symbols in the query sequence, leading to an increase in the number of matching positions on the reference genome. Moreover, the size of the output set will generally increase with more wildcards impacting the time needed to write things to the output device (see above).

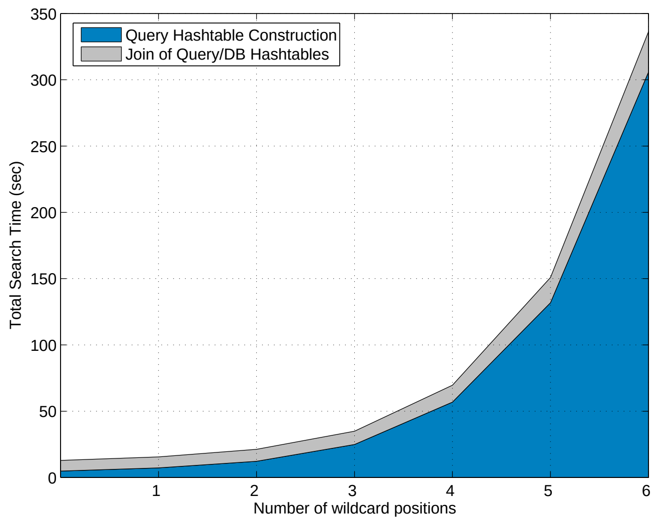


**Figure 15.** Runtime vs Wildcard Cardinality: Search on Chromosome 1 for one Million queries using `Q-Pick` . Total search time is reported with respect to cardinality of wildcard positions.

The results for this experiment are shown in Figure 15 for different number of wildcards (zero to six positions). Matches for one million distinct query sequences from the *mouse* genome are sought in the forward strand of chromosome 1 of the *human* genome. As expected, there is an exponential increase in the runtime, with the majority of the time being consumed by the query hashtable generation, while the search over the corresponding bins of the query and the database tables remains approximately constant.

For fewer than 4 wildcard positions the search time is less than 1 minute given 1 million distinct queries. Notice, that these numbers still indicate the great improvement in performance that can be achieved by `Q-Pick` , since 4 wildcards on queries with approximate length of 20, represent uncertainty over $20\%$ of the total number of positions in the query.

## 3.4 Scale-up experiment: Full Genome Search

In our final experiment, we seeks instances of short, variable-length queries in the entire human genome, looking for matching positions on both forward and reverse strands. Here we demonstrate that our technique can process millions of distinct queries against the human genome and retrieve results in just minutes on a single processor.

Figure 17 shows the pre-processing that is required for the target genome hashtable generation. Building the hashtable for all 24 chromosomes of the human genome requires a little more than 7 hours. Note that this is a one-time, offline cost. The memory requirements for creating the hashtable for human chromosome 1 (the longest chromosome) are 2.1GB, therefore a computer with 3GB of RAM is sufficient for executing full genome search. The lengths of the all human chromosomes used in this experiment are summarized in Figure 16.

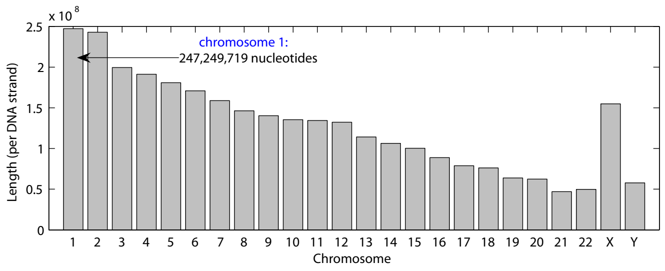


**Figure 16.** Lengths (in nucleotides) of the human chromosomes utilized in our experiments.

The query hashtables are generated from the query set and have to be regenerated every time, for a low runtime cost of ~10 seconds for 1 million query sequences; for 10 million query sequences the time needed to build the table rises to ~50 seconds. The mix of queries was created by extracting subsequences between 17 and 30 letters from the mouse genome and randomly replacing 1-3 positions by wildcards. Indicatively, to generate the hashtable for 1 million queries and their reverse complement sequences with 2 wildcards requires 1.6GB of RAM. We measure the time of two subtasks:

1. The time spent on generating the query database hashtable
2. The search time (or join) of the hashtable bins of the queries with the corresponding database bins (genome).

These runtime costs are reported in Figure 18. One can observe that for 10 distinct million queries the response time is less than 300 sec. This is a very important result, since it allows for a very fast turnaround time, between query set generation from various sources and discovery of their positions on the genome within minutes.

Finally, we also stress that these results correspond to experiments performed on a single processor. Since a separate hashtable is created for each chromosome, the search process of `Q-Pick` can be trivially parallelized, by distributing portions of different chromosome hashtables to different host systems, while inducing minimal communication costs. In the future, we plan to evaluate the benefits of offering a parallelized version of the algorithm. However, we should stress that one of the main attractions of our methodology is its exceptional performance even on a single processor.

## 4. CONCLUSIONS

In this work we presented `Q-Pick` , a new methodology for anchoring sequence fragments on a genome and reporting all their matches. `Q-Pick` is considerably faster than the other currently available schemes, while at the same guaranteeing (unlike some of the other methods) the reporting of all the locations where the queries at hand can be anchored. Our experiments on the human genome indicate that `Q-Pick` can be *up to 60 times* faster than state-of-the-art short sequence matching techniques. Our methodology achieves its exceptional performance through a host of features, such as data packetization, simplicity of implementation, fast hashtable joins and intelligent data pruning.

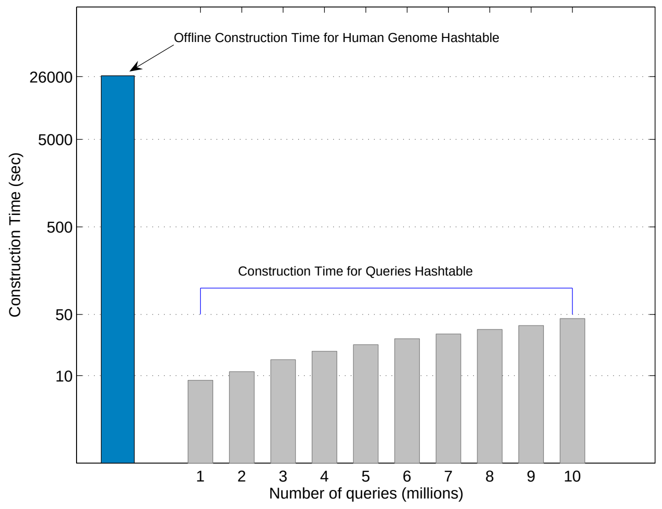


**Figure 17.** Time to generate the necessary hashtables (log scale on time). The left-side bar shows the time to generate the full human genome hashtable, which represents a one-time, offline cost. The rightmost bars demonstrate the time to generate the query hashtables, which need to be regenerated for different query sets.

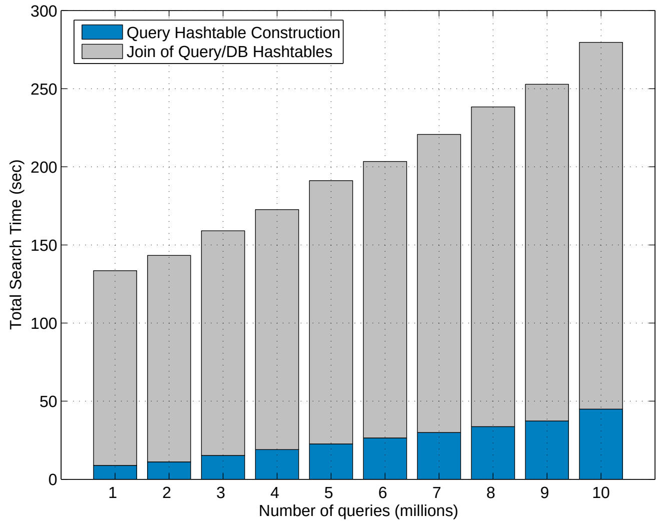


**Figure 18.** Search time against whole human genome (both forward and reverse DNA strands) for increasing query set cardinalities.

## 5. REFERENCES

[1] `http://seqanswers.com/forums/showthread.php?t=145`.

[2] `http://www.sanger.ac.uk/Users/lh3/seq-nt.html`.

[3] N. Cloonan, A. Forrest, G. Kolle, et al. Stem cell transcriptome profiling via massive-scale mRNA sequencing. In *Nature Methods 5 (7)*, pages 613–619, 2008.

[4] A. J. Cox. (Unpublished) ELAND: Efficient Large-Scale Alignment of Nucleotide Databases. Illumina, 2008.

[5] J. Deng, R. Shoemaker, and B. Xie. Targeted bisulfite sequencing reveals changes in DNA methylation associated with nuclear reprogramming . In *Nature Biotechnology 27*, pages 353–360, 2009.

[6] J. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. In *Genome Research 17 (11)*, pages 1697–1706, 2007.

[7] S. Elbashir, J. Harborth, W. Lendeckel, A. Yalcin, K. Weber, and T. Tuschl. Duplexes of 21-nucleotide RNAs mediate RNA

interference in cultured mammalian cells. In *Nature 411*, pages 494–498, 2001.

[8] R. Fernandes and S. Skiena. MultiPrimer: a system for microarray PCR primer design. In *Methods Mol Biol., 402*, pages 305–314, 2007.

[9] L. Gasieniec, C. Li, P. Sant, and P. Wong. Randomized probe selection algorithm for microarray design. In *J Theor Biol, 248(3)*, pages 512–521, 2007.

[10] C. Iseli, G. Ambrosini, P. Bucher, and C. Jongeneel. Indexing strategies for rapid searches of short words in genome sequences. In *PLoS ONE, 2(6):e57*, 2007.

[11] W. Kent. BLAT–the BLAST-like alignment tool. In *Genome Research 12(4)*, pages 656–664, 2002.

[12] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. In *Genome Biology 10:R25*, 2009.

[13] L. Laurent, E. Wong, G. Li, T. Huynh, A. Tsirigos, et al. Dynamic Changes in the Human Methylome During Differentiation. In *Genome Research*, 2010, In Press.

[14] T. J. Ley, E. R. Mardis, et al. DNA sequencing of a cytogenetically normal acute myeloid leukaemia genome. In *Nature 456 (7218)*, pages 66–72, 2008.

[15] H. Li. MAQ: Mapping and Assembly with Quality. In `http://maq.sourceforge.net/`.

[16] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. In *Bioinformatics 25*, pages 1754–1760, 2009.

[17] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. In *Genome Research 18*, pages 1851–1858, 2008.

[18] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. In *Bioinformatics Note, 24(5)*, pages 713–714, 2008.

[19] W. Li and X. Ying. Mprobe 2.0: computer-aided probe design for oligonucleotide microarray. In *Appl Bioinformatics 5(3)*, pages 181–186, 2006.

[20] X. Li, Z. He, and J. Zhou. Selection of optimal oligonucleotide probes for microarrays using multiple criteria, global alignment and parameter estimation. In *Nucleic Acids Res., 33(19)*, pages 6114–6123, 2005.

[21] R. Lister, M. Pelizzola, and R. Dowen. Human DNA methylomes at base resolution show widespread epigenomic differences. In *Nature 462*, pages 315–322, 2009.

[22] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD*, pages 247–258, 1996.

[23] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proc. of ACM SIGMOD*, pages 833–844, 2007.

[24] E. D. Pleasance, R. K. Cheetham, et al. A comprehensive catalogue of somatic mutations from a human cancer genome. In *Nature, doi:10.1038/nature08658*, 2009.

[25] E. D. Pleasance, P. J. Stephens, et al. A small-cell lung cancer genome with complex signatures of tobacco exposure. In *Nature, doi:10.1038/nature08629*, 2009.

[26] I. Rigoutsos, T. Huynh, K. Miranda, A. Tsirigos, A. McHardy, and D. Platt. Short blocks from the non-coding parts of the human genome have instances within nearly all known genes and relate to biological processes. In *PNAS, 103(17)*, pages 6605–6610, 2006.

[27] S. Rumble and M. Brudno. SHRiMP: SHort Read Mapping Package. In `http://compbio.cs.toronto.edu/shrimp/`, University of Toronto.

[28] O. Snøve and T. Holen. Many commonly used siRNAs risk off-target activity. In *Biochem Biophys Res Commun., 319(1)*, pages 256–263, 2004.

[29] J. Stenberg, M. Nilsson, and U. Landegren. ProbeMaker: an extensible framework for design of sets of oligonucleotide probes. In *BMC Bioinformatics, 6:229*, 2005.

[30] C. Trapnell and S. L. Salzberg. How to map billions of short reads onto genomes. In *Nature Biotechnology 27*, pages 455–457, 2009.

[31] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. In *Nature Reviews Genetics 10 (1)*, pages 57–63, 2009.

[32] R. Wernersson, A. Juncker, and H. Nielsen. Probe selection for DNA microarrays using OligoWiz. In *Nat Protoc, 2(11)*, pages 2677–2691, 2007.

[33] T. Yamada and S. Mirishita. Accelerated off-target search algorithm for siRNA. In *Bioinformatics, 21(8)*, pages 1316–1324, 2005.

[34] D. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. In *Genome Research 18 (5)*, pages 821–829, 2008.