BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 18 2014, pages 2543–2550
doi:10.1093/bioinformatics/btu330

Genome analysis

Advance Access publication May 27, 2014

CWig: compressed representation of Wiggle/BedGraph format

Do Huy Hoang[1] and Wing-Kin Sung[1,2,*]

[1]Department of Computational and Systems Biology, Genome Institute of Singapore, Singapore 138672 and [2]Department of Computer Science, School of Computing, National University of Singapore, Singapore 117417

Associate Editor: Inanc Birol

ABSTRACT

Motivation: BigWig, a format to represent read density data, is one of the most popular data types. They can represent the peak intensity in ChIP-seq, the transcript expression in RNA-seq, the copy number variation in whole genome sequencing, etc. UCSC Encode project uses the bigWig format heavily for storage and visualization. Of 5.2 TB Encode hg19 database, 1.6 TB (31% of the total space) is used to store bigWig files. BigWig format not only saves a lot of space but also supports fast queries that are crucial for interactive analysis and browsing. In our benchmark, bigWig often has similar size to the gzipped raw data, while is still able to support ~5000 random queries per second.

Results: Although bigWig is good enough at the moment, both storage space and query time are expected to become limited when sequencing gets cheaper. This article describes a new method to store density data named CWig. The format uses on average one-third of the size of existing bigWig files and improves random query speed up to 100 times.

Availability and implementation: http://genome.ddns.comp.nus.edu.sg/~cwig

Contact: ksung@comp.nus.edu.sg

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on March 3, 2014; revised on May 1, 2014; accepted on May 5, 2014

1 INTRODUCTION

As the next-generation sequencing (NGS) cost reduces, huge amount of reads can be generated nowadays. After aligning the reads on a reference genome, we can generate the read density, i.e. the number of NGS reads covering each base in the genome. Density data are useful because it can be used to represent the transcript expression in RNA-seq (Hu *et al.*, 2013), the peak intensity in ChIP-seq (Liu *et al.*, 2011), the copy number variation in whole genome sequencing (Bock, 2012), etc. For example, Figure 1 shows plots of density signals of a ChIP-seq region and a RNA-seq region, respectively.

Currently, read density is often represented using the wiggle (wig) format, the bedGraph format or the bigWig format. They all store the densities of NGS reads along the whole reference genome. Wig and bedGraph are uncompressed text formats, thus, are usually huge. BigWig (Kent *et al.*, 2010) is the compressed form of wig and bedGraph. Its compression approach is to sort and partition the density data into blocks and compress them by gzip. BigWig also supports a few types of queries over any selected region: coverage, max, min, average and standard deviation. These queries facilitate efficient downstream analysis and enable fast visualization of the data.

With bigWig format, UCSC genome browser (Karolchik *et al.*, 2014) can support interactive browsing of density data. In fact, bigWig is one of the most popular track types. In the hg19 browser, ~4400 tracks (10% of all hg19 tracks) are bigWig tracks, and they use 1.6TB (it is equivalent to 31% of the total space for all UCSC hg19 tracks). To reduce space and improve query speed, the resolution of the density signals of some UCSC tracks has been reduced, which affects the accuracy. In the future, it is important to reduce the storage space of density data and improve their query speed while maintaining the accuracy of the data.

Our project aims to develop an alternative storage format for density signal. Our design is based on careful observations of the data and knowledge of succinct and compressed data structures. For example, we observed that mapping locations of NGS are usually overlapped. Regions with non-zero intensity are often clustered. This fact enables us to reduce the space. Another observation is that the density values of adjacent regions are not independent. Storing the differences between adjacent density values can reduce the size of 80% of the datasets in UCSC hg19. To enable fast queries, we use data structures like SDArray (Okanohara and Sadakane, 2007) that can compress data while still allowing random access. We also adopt a modified Cartesian tree (Fritz *et al.*, 2011) that uses linear number of bits and provides constant time min/max query.

Similar to UCSC bigWig tool, cWig tool also implements the remote file access feature. In this feature, the program and the data file can be placed in different computers. The program can answer queries by accessing the data file through the HTTP/HTTPS network protocol.

In our experiment using all UCSC hg19 database, the cWig format uses on average one-third of the size of existing bigWig files, and uses much lower space in high resolution data files. In addition, it also improves query speed by 10–100 times depending on the query types.

2 BACKGROUNDS

UCSC database stores and displays many types of genome-related data. They can generally be divided into three groups of formats. Sequence formats: store raw DNA sequences and quality scores. Examples include SAM/BAM (Li *et al.*, 2009), FASTQ (Cock *et al.*, 2010) formats. Annotation formats: store information about some biological features (e.g. genes, variants)

[*]To whom correspondence should be addressed.

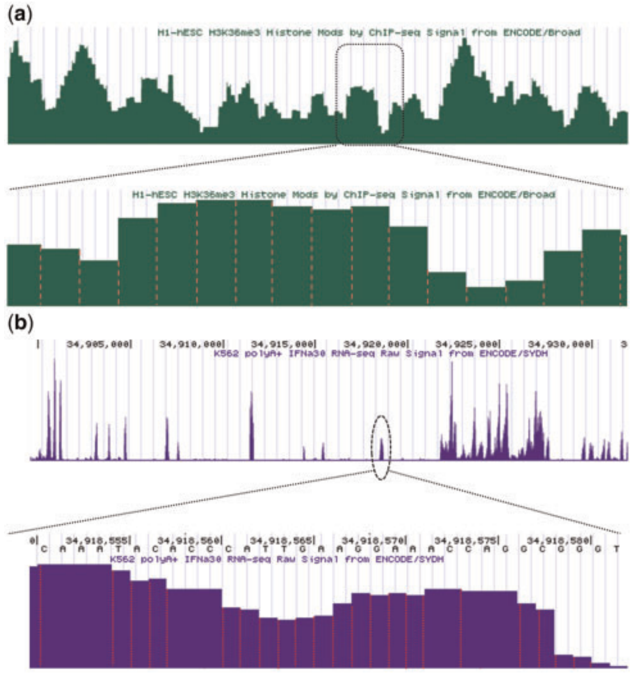


Fig. 1. (**a**) A zoom region in ChIP-seq file. (**b**) A zoomed region in a RNA-seq file. The dotted lines indicate boundaries of two consecutive intervals

located in a genome. Some popular formats in UCSC are Bed, BigBed (Kent *et al.*, 2010) and VCF (Danecek *et al.*, 2011). Some annotation formats are designed to keep different types of features, for example, (Hoffman *et al.*, 2010) and (Gundersen *et al.*, 2011). Signal formats: store continuous numerical signal values for each genome bases. Examples include Wiggle, BedGraph and bigWig formats.

The sequence and annotation files can be big, but they only require simple queries, i.e. list or count all sequences/annotations in a given region. This query can be solved by adding some index pointers on top of the existing formats. The signal files are structurally simple; however, it requires fast summary operations over some long regions.

This article focuses on improving the existing signal formats. The raw density dataset is usually big (measured in Giga bytes per file) and contains a lot of duplicated information. To reduce size, bigWig applies the following compression scheme. It keeps a set of non-overlapping intervals such that the bases in each interval share the same signal value. Intervals with zero intensity or missing values are usually omitted. All intervals are sorted by their starting positions and they are partitioned into blocks of 512 by default. Each block of intervals and their corresponding signal values are compressed using the gzip algorithm in zlib library. To allow partial random access, bigWig stores the starting locations of all blocks using an R-tree-based index (Guttman, 1984), which is commonly used for geographical data.

In addition to the original data, bigWig also stores extra tables to provide fast computation of four summary operations over any query interval. These operations are mean, min/max, coverage and standard deviation. They are crucial for UCSC genome browser visualization function.

Before we formally define the four operations, we need some additional notations. Let r_k be the value at position k of the genome. If there is no value at position k, we denote r_k as $\mathtt{NaN}$. Operations that involve $\mathtt{NaN}$ are $\mathtt{NaN}+\mathrm{x}=\mathrm{x}$, $\mathtt{NaN}\cdot\mathrm{x}=\mathrm{x}$, $1/0=\mathtt{NaN}$, $\min(\mathtt{NaN},\mathrm{x})=\mathrm{x}$, $\max(\mathtt{NaN},\mathrm{x})=\mathrm{x}$, where x is any value (including $\mathtt{NaN}$). For any query range $p..q$, let N be the number of positions k in $p..q$, where $r_k\neq\mathtt{NaN}$. The four operations are defined as follows.

- *coverage*(p,q): Proportion of positions k where $r_k\neq\mathtt{NaN}$, that is, $N/(q-p+1)$.
- *mean*(p,q): The arithmetic mean of the non-NaN values in $p..q$, that is, $\frac{1}{N}\sum_{k=p}^{q}r_k$.
- *min_val*(p,q) and *max_val*(p,q): the minimum/maximum value in $p..q$, that is, $\min_{k=p..q}\{r_k\}$ and $\max_{k=p..q}\{r_k\}$.
- *stdev*(p,q): The standard deviation of the non-NaN values in $p..q$, that is, $\sqrt{\frac{1}{N}\left(\sum_{k=p}^{q}r_k^2\right)-mean(p,q)^2}$.

The extra tables in bigWig file stores precomputed answers of the operations in different zoom levels. For example, zoom level 1 stores answers for regions of length 50 000 bases and zoom level 2 stores answers for regions of length 5000 bases. The precomputed tables are also indexed using R-trees.

3 OBSERVATIONS

This section describes our observations on the bigWig data in UCSC hg19 database. bigWig groups bases that have the same values into intervals instead of storing signal values for each individual base. The problem becomes storing a set of tuples, i.e. (s_i,e_i,v_i) where, s_i and e_i are the start and the end positions of the intervals in a genome; and v_i is the signal value of the bases in the interval $s_i..e_i$. As the positions and the values are highly independent across the database, we study them separately in the next two subsections.

3.1 Observations on interval positions

This section discusses our observations on the characteristics of the interval data $s_i..e_i$ stored in bigWig format. For high-density regions, NGS reads are often overlapped. Once the reads are piled up to generate the coverage data, each high-density region is expected to form a set of consecutive intervals. To illustrate, Figure 1 shows the density plots of a ChIP-seq region and a RNA-seq region. In both data types, we observed that the position intervals are usually consecutive (i.e. the start of the next interval equals the end of the previous one).

To precisely measure this characteristic, we define a measurement called consecutiveness, which is the percentage of intervals in a signal data file that have their start positions equal the end positions of their adjacent intervals. The consecutiveness is zero when no interval stays next to another. It approaches one when all intervals are chained together.

Figure 2a plots the proportion of bigWig files in UCSC hg19 database based on consecutiveness. We found that 81% of the files have the consecutiveness >0.5. To have a clear picture, Figure 2b further shows the relationship between the consecutiveness and the coverage. (Recall that the coverage is the

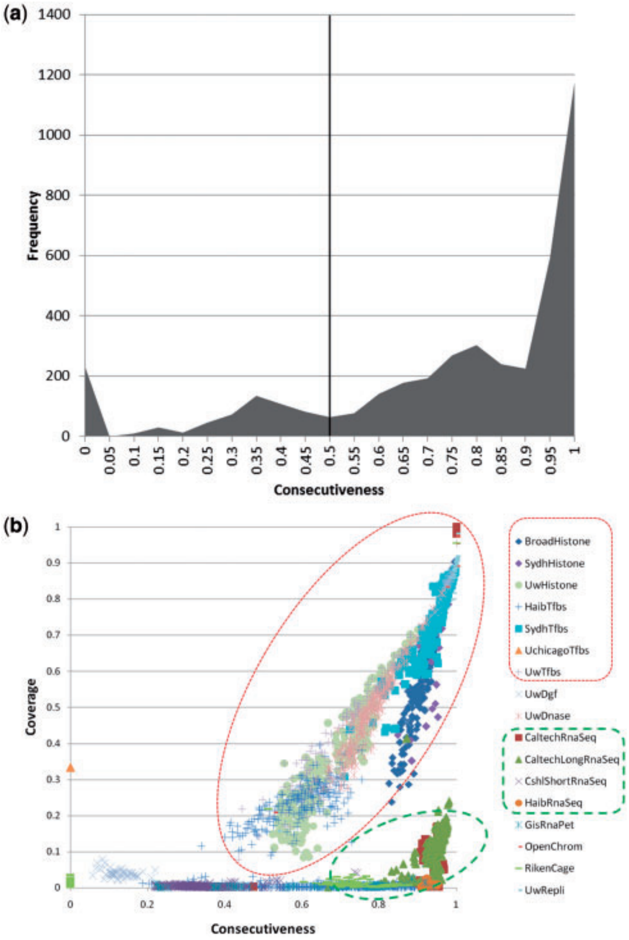


Fig. 2. (**a**) Histogram of the datasets based on consecutiveness. (**b**) Coverage versus consecutiveness in UCSC hg19 bigWig files. The big dotted oval highlights most Chip-seq datasets. The small oval highlights low coverage, but high consecutive datasets

percentage of bases of the genome that have signal data.) Intuitively, we expect high coverage files have high consecutiveness. This is actually true as shown in the figure. Most of the Chip-seq data files (highlighted in red oval) are high in both coverage and consecutiveness. However, many RNA-seq files only have high consecutiveness. That means high consecutiveness may be a characteristic of RNA-seq data. Section 4.2 will use this property to reduce the space consumption for storing the positions of the intervals.

3.2 Observations on signal values

This section discusses our observations on signal values in bigWig files. Let v_i be the signal value of an interval $s_i..e_i$. Figure 1 shows that signal values of adjacent intervals are similar for most cases. We suspect that storing the differences (i.e. $v_{i+1} - v_i$) may be better than storing the raw signal values (i.e. v_i). To validate this observation, we compare the entropy of raw signals and the entropy of signal differences of adjacent intervals. [Under certain conditions, entropy (Cover and Thomas, 1991) is the minimum number of bits required to store each element in a sequence of values.]

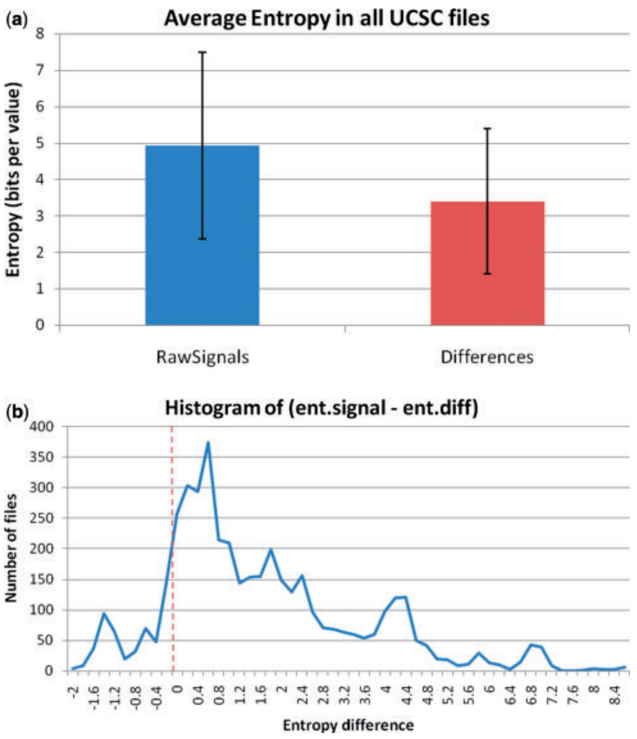


Fig. 3. (**a**) Average entropy of raw signals and their differences in UCSC files. (**b**) Histogram of the entropy of the values minus the entropy of the differences in each file

Figure 3 shows that, among all UCSC bigWig files, the average entropy of raw signals is ~4.9 bits, whereas the entropy of differences is around 3.2 bits. This means that, with a suitable compression scheme, storing differences uses less space than storing raw signal values on average.

To be more precise, we try to find the list of bigWig tracks, where storing differences is better by computing the discrepancy between the two entropies for each bigWig track. Figure 3b shows the histogram plot of the results. We found that 81% of the bigWig tracks (represented by the area under the curve on the right side of the zero line) give smaller entropy when the differences of the adjacent signals values are stored. In other words, we can classify the files into two classes. The first class is smaller by storing differences of the signals. The second class is smaller by storing raw signal values.

Our second observation is that certain signal (or difference) values occur more frequently in the bigWig file. To be precise, we define the number of frequent signal (or difference) values in a bigWig file as the minimum number of distinct values whose sum of occurrences makes up 75% of the total number of values in that file. Figure 4 shows the number of bigWig files that have x frequent signal (or difference) values for all x. Of 4400 bigWig files in UCSC hg19, about 1500 files have less than six frequent raw signal values, and ~2500 files have less than six frequent differences values. Most of the files have <60 differences values.

We further investigate the distributions of the values in each file. After studying many examples, we found that the frequent

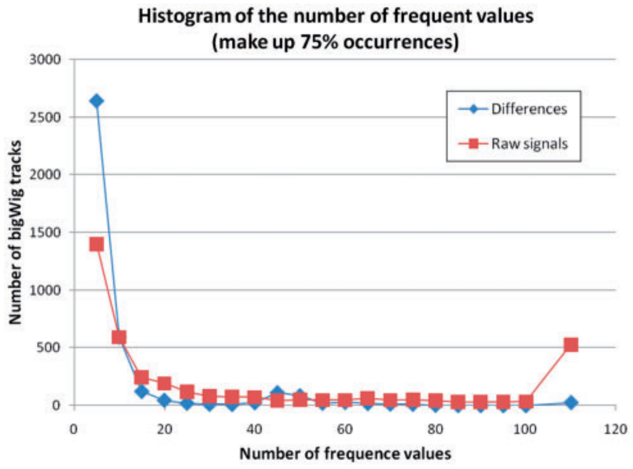


Fig. 4. Histogram of frequent values

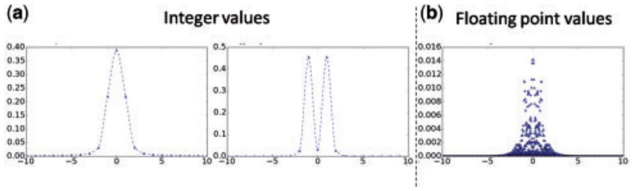


Fig. 5. Histogram of signal differences. (**a**) and (**b**) are two common histograms observed in integer value signal files. (**c**) is a common histogram observed in floating point value signal files. In these subfigures, the *X*-axes are the difference between adjacent signal values shown from –10 to 10. *Y*-axes show the frequencies of these signal differences. The maximal frequencies shown in (a), (b) and (c) are 0.4, 0.5 and 0.016, respectively

values are usually close to zero. For integer signal files, Figure 5a and b show the typical distributions of signal differences. They usually contain one or two peaks in the center. For floating point signal files, Figure 5c shows the typical distribution of the signal differences. They often have dense values near zero. We ran a simple classifier on the database and found that of 2627 integer signal files, 1851 files have two peak shape that look like Figure 5b, whereas 813 files have shape that are similar to Figure 5a.

In summary, we have three observations for the signal values. More than half of the data is better stored by differences. Most data files have a small set of frequent values. The frequent values are usually small and close to zero. We will use these observations to design schemes for storing signal values.

4 METHODS

Using the knowledge from the observations of all bigWig files in UCSC hg19 database, this section presents our storage scheme.

4.1 SDArray

One of the frequently used components in our design is SDArray proposed by (Okanohara and Sadakane, 2007). It can be seen as a compressed array of increasing integers. We use this data structure for storing both data and index pointers. The advantages of this data structure over the traditional search tree is that, it uses nearly optimal number of bits while still provides $O(1)$ time to access and less than $O(\log_2 m)$ time to search (where m is the number of elements). There are a few alternative compressed structures, which have similar properties as described by Raman *et al.*, 2002 and Patrascu, 2008. SDArray is used because of its speed and simplicity. In addition, it has good compression ratio when the values are not dense, which is commonly observed in our data.

The details of SDArray are as follows. Consider an array of non-decreasing nonnegative integers $P[1..m]$. Storing $P[1..m]$ explicitly costs $m\lceil\log_2 n\rceil$ bits (where n is the biggest number). SDArray is a compressed data structure storing the array $P[1..m]$ and enables constant time access of any element $P[i]$. It also provides an operation called $rank(P, x)$ to find the first element $P[i]$ that is greater than or equal to x, i.e. $rank(P, x) = \min\{i\,|\,P[i] \geq x, i \in 1..m\}$. Let $n = P[m]$. The SDArray for the array P uses $1.56m + m\log_2(n/m) + o(m)$ bits and computes *rank* operation in $O(\log_2(\min(n/m, m)))$ time. This data structure is better than explicit storage when $n \gg m$ and $m > 4$.

4.2 Compression schemes for interval positions

Consider a set of m position intervals $\{s_i..e_i | i = 1..m\}$. Without loss of generality, assume the intervals are sorted in increasing order of s_i. This section describes two alternative schemes (basic scheme and space saving scheme) to store the position intervals. Our two schemes also support random access of the values s_i and e_i. To implement compatible bigWig operations, our schemes require an operation called *find_interval*(p) that finds the maximal index i, such that $s_i \leq p$, and an operation called *cover_len*(k) that reports the total length of the first k intervals (i.e. $\sum_{i=1}^{k}(e_i - s_i + 1)$).

The basic scheme has better access time for the queries, whereas the space saving scheme is more compact when there are many consecutive intervals. CWig uses the space saving scheme, if the consecutiveness (defined in the observation section) is >0.5; otherwise, it uses the basic scheme.

Basic scheme: The basic scheme stores the starting positions and interval lengths in two SDArrays: $S[1..m]$ and $L[1..m+1]$, respectively, such that $S[i] = s_i$, $L[0] = 0$ and $L[i] = \sum_{k=1}^{i-1}(e_k - s_k)$. Given S and L, s_i and e_i equal $S[i]$ and $S[i] + L[i+1] - L[i]$, respectively. Operation *find_interval(p)* equals $rank(S, p)$. Operation *cover_len(k)* equals the value of the k-th entry of L plus k. Hence, all operations take $O(\log_2(n/m))$ time.

The space complexity for this scheme is $m(3.12 + \log_2(n/m) + \log_2(l/m)) + o(m)$ bits, where $n = s_m$, and l is the total length of all intervals (i.e. $L[m]$). This scheme enables efficient query. It also has good space usage when the intervals are sparse (e.g. in RNA-seq datasets).

Space saving scheme: By the observations in the previous section, the space saving scheme groups the consecutive intervals into segments to save space. Precisely, we group consecutive intervals $(s_i, e_i), \ldots, (s_j, e_j)$ into one segment, if $e_k = s_{k+1}$ for $k = i, .., j-1$. The space saving scheme stores the starting positions of segments, the numbers of intervals in each segment and the lengths of all intervals. Assume that there are g segments, we store:

- $G[1..g]$ is a length-g array, where $G[j]$ is the start position of the j-th segment.
- $I_c[1..g+1]$ is an array such that $(I_c[i+1] - I_c[i])$ equals the number of intervals in the i-th segment.
- $L[1..m+1]$ contains the prefix sum of the lengths (same as the one in basic scheme).

To find the start of the interval i (i.e. the value of s_i), we first compute the segment j that contains the interval i by calculating $j = rank(I_c, i)$, then $s_i = G[j] + L[i] - L[I_c[j]]$. The end of the interval, $e_i = s_i + L[i+1] - L[i]$.

```
function $find\_interval\ p$
    $j = rank(G, p)$
    $i = rank(L, (p - G[j]) + L[I_c[j]])$
    if $(i < I_c[j+1])$ then return $i$
    else return $L[I_c[j+1]]$
```

The operation *find_interval(p)* can be computed using a two-step algorithm. The first step finds the segment nearest to p. Because the intervals inside each segment are consecutive, the second step finds the index of the interval that contains p, using the distance between p and the start of the segment. The operation *cover_len(k)* equals the value of the k-th entry of L plus k.

The space complexity for this scheme is $1.56m + m\log_2(l/m) + g(3.12 + \log_2(n/g) + \log_2(m/g)) + o(g + m)$ where l is the total length of the intervals, g is the number of groups and m is the number of intervals. The estimated space requirement is better than the basic scheme when $2g < m$. That is when each group on average has more than two intervals (i.e. the consecutiveness is >0.5).

4.3 Compression schemes for signal values

By the observations in Section 3.2, we design our compression scheme for storing values and the auxiliary data structure to support the required query.

The compression has two main stages. The first stage converts the signals into integers and decides whether we need to store the raw signal values or the differences based on the entropy. It also applies some common transformations to make numbers easier for compression. The second stage uses a mixture of methods to compress the integers.

Transformations: Let $V = \{v_1, v_2, \ldots, v_m\}$ denote the signal values. For floating point datasets, we convert all signal values into integers by multiplying with a scale factor. Precisely, we scan all values in V and identify the maximum number of digits α after the decimal point; then, every value is multiplied by the same scaling factor $f = 10^{\alpha}$. For practical purpose, we keep at most seven fractional decimal digits of precision, which is compatible to the precision level in bigWig format. It is similar to use IEEE's 32-bit floating point numbers for storing signal values.

The next step is to decide whether we store the signal values or differences. To make the decision, we compute the entropy of the values and the differences. If the entropy of the values is smaller, we will store the set $B = \{b_i\}$ such that $b_i = v_i f$ for $i = 1..n$ where, f is the scaling factor. Otherwise, we store the set $B = \{b_i\}$ such that $b_i = (v_{i+1} - v_i)f$ for $i = 1..n-1$. To avoid the gaps between the numbers introduced by the scaling, we convert B into C such that c_i equals the rank of the values of b_i in sorted order.

Compression: The previous section showed that only a few signal differences have high frequency. Furthermore, many signal differences with high frequency are scattered around zero. To capture this type of distributions, we use two compression methods: Huffman code and Elias delta code. Each method has its own strength and weakness.

Elias delta code (Elias, 1975) is a variable length encoding scheme for positive integers. It represents an integer x in $\lfloor \log x \rfloor + 2\lfloor \log_2 \lfloor \log_2 x + 1 \rfloor \rfloor + 1$ bits. This compression scheme is asymptotically optimal when the numbers are uniformly random in a large range.

Huffman code (Huffman, 1952) is a variable length encoding scheme for a set of symbols (i.e. characters). It encodes each symbol by a new sequence of bits. This compression wastes at most 1 bit per symbol when the probability distribution is known. However, because it needs to store a symbol mapping table, the method is not practical when the number of symbols is large.

To encode the set of numbers C from the transformation stage, we use Huffman code to capture the small set of frequent numbers and use Elias delta code for the rest. The details are as follows. We construct a Huffman code with 128 symbols. The most frequent 127 values in C are encoded by 127 Huffman symbols. The remaining values share the 128th Huffman symbol as their prefix and use the delta code values as suffixes. The weights used to build the Huffman symbols are the frequencies of the values. Note that we choose 128 symbols because Figure 4 showed that most of the files have <100 frequent values.

The signal values V is, therefore, represented by storing the value C, and necessary information to reverse transform from values C to values V (e.g. the factor f, the scheme is raw values or differences, the ranks, the Huffman code table).

Auxiliary data structures for queries: We also require a few additional auxiliary data structures and intermediate operations to implement the summary operations defined in Section 2 (i.e. min/max, average and SD).

To support the min and max operations, we use Cartesian tree from (Fritz *et al.*, 2011). This structure uses $2m + o(m)$ bits. It supports computation of the minimum/maximum values in any range using $O(1)$ time. Formally, the data structure provides two operations $min\_idx(i, j) = \arg\ \min_{k \in i..j}\{v_k\}$ and $max\_idx(i, j) = \arg\ \max_{k \in i..j}\{v_k\}$.

For the average and SD operations, we need auxiliary data structures to compute two intermediate operations: sum and square sum of the values. The intermediate operations are defined as follows: $cover\_val(k) = \sum_{j=1}^{k}(e_j - s_j + 1)v_j$ and $cover\_val\_sqr(k) = \sum_{j=1}^{k}(e_j - s_j + 1)v_j^2$ for $k = 1, \ldots, m$. To implement operations *cover_val* and *cover_val_sqr*, we keep one sampled value in every 64 values of the functions. The sampled values are stored in SDArray for fast access. To compute the values that are not sampled, we jump to the nearest sampled value and sequentially extract (s_j, e_j, v_j) to compute the exact sum.

4.4 Query

Previous subsections have outlined our storing scheme for the positions and values of the intervals. This section shows how to use these components to support the four summary query operations defined in Section 2. In general, given a query region $p..q$, the query asks for some summary values (e.g. average, min/max, SD, coverage) of the signal values of the genome positions from p to q. The details are as follows.

Coverage query: Given the input region $p..q$, the coverage query *coverage(p,q)* computes the proportion of non-NaN bases. Note that the number of non-NaN bases, which equals $\frac{1}{(q-p+1)}(q \cdot coverage(0, q) - (p-1)coverage(0, p-1))$. Let j be the largest index such that s_j is less than or equal to q (i.e. $j = find\_interval(q)$). We have $q \cdot coverage(0, q) = cover\_len(j) - \min\{e_j - s_j, q - s_j\} + 1$. Similarly, we can compute $(p-1) \cdot coverage(0, p-1)$ using $find\_interval(p-1)$ and the interval values.

Min/max query: The minimum/maximum of signal values in a query region $p..q$ can be computed in three steps. First, we find the set of intervals $\{(s_i, e_i), \ldots, (s_j, e_j)\}$ that overlap with the query region $p..q$. This can be done by computing $find\_interval(p)$ and $find\_interval(q)$. The second step uses operations $min\_idx(i, j)$ or $max\_idx(i, j)$ to find the index of the minimal/maximal value in constant time. The last step extracts the actual signal values.

Mean query: $mean(p, q) = \frac{1}{n}\sum_{k=p}^{q} r_k$ where r_i is the value of the i-th base, and n is the number of non-NaN bases, i.e. $n = (q - p + 1)coverage(p, q)$. Note that $\sum_{k=p}^{q} r_k = \sum_{k=0}^{q} r_k - \sum_{k=0}^{p-1} r_k$. The value of $\sum_{k=0}^{q} r_k$ can be computed by (1) let $j = find\_interval(q)$ and (2) $\sum_{k=0}^{q} r_k = cover\_val(j) + v_j(\min\{e_j - s_j, q - s_i\} + 1)$. Similarly, we can compute $\sum_{k=0}^{p-1} r_k$.

Standard deviation query: $stdev(p, q)$ can be computed using the formula $\sqrt{\frac{1}{n}\sum_{i=p}^{q} r_i^2 - \frac{1}{n^2}mean(p, q)^2}$, where, r_i and n are defined same as above. Using similar approach as the mean query, the sum of squared signal values $\frac{1}{n}\sum_{i=p}^{q} r_i^2$ can be computed from the intermediate queries *cover_sum_sqr* and *find_interval*.

4.5 Remote file access

Our solution for remote access feature is to use a simple network layer that handles HTTP 1.1 byte ranges and keep-alive protocols. Once a data file is placed under a web server that supports the HTTP protocol (e.g. Apache, Microsoft IIS and nginx), it can be queried from different computer to get any block of data. The implementation also supports HTTPS protocol if OpenSSL library is available.

To avoid duplicated data transfer and network protocol overhead, a simple file caching scheme is implemented. Any data requested over the network is read in blocks of 16 KB and stored in a cache file. An additional bit-map file is kept to mark down blocks that have been saved locally. Multiple queries to some close locations are likely to access the same data block, hence, do not incur new network request. In addition, the overhead to start transferring data over the network is high (e.g. in milliseconds). It is more beneficial to transfer data in blocks.

To enhance the performance of block transferring and file caching, cWig reorganizes the component data structures to make data access localized. It groups small, fixed size and frequently accessed fields of different data structures into a consecutive segment called ‘control segment’. (The segment usually stores the length, counter and metadata of the data structures.) The large and variable length data are stored in another segment of the file. When the data structure is loaded remotely, the data in the control segment is more likely to be transferred in one request and cached; therefore, it helps to reduce the delay between queries.

5 EXPERIMENT RESULTS

In this section, we present three sets of experiments. The first set of experiments compares the sizes of bigWig and cWig files. It also compares different alternatives of our design to support our final choice. The second set of experiments compares the speed between bigWig and cWig in one machine. The last set of experiments compares the remote query speed of cWig’s and bigWig’s tools.

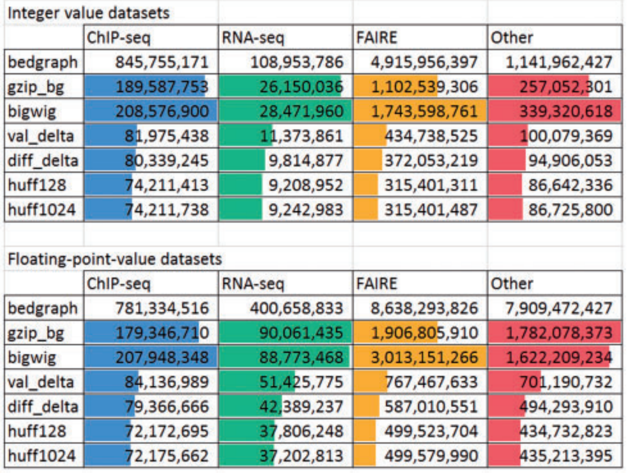
We use three datasets for the experiments: full dataset for size measurement, sampled dataset for the speed measurement on one computer and a few selected files for the remote access experiments.

The full dataset consists of all bigWig files in UCSC hg19 database (~4400 files). The UCSC bigWig files use a total of 1.6 Terabytes. To have a clear picture, we categorize the files in UCSC into groups by value types (i.e. integer signal versus floating point signal) and by data types (i.e. ChIP-seq, RNA-seq, DNAse, FAIRE and Other). This dataset is used in the section on file size comparison.

The sampled dataset is a subset of the full dataset. The files are grouped similarly as the full dataset. However, each group only contains 5–10 sampled files. (The detailed list of files can be found in the Supplementary C.) The sampled datasets are used for running time comparison.

Furthermore, three files from UCSC hg19 of different sizes are selected for the remote query speed experiments.

Note that the name bigWig, cWig or gzip is used to refer to both the file format and tool/program to access the format. For bigWig, there are a few tools that can create, extract and randomly access the format. We use the latest version of the tool provided by the original authors (in Kent *et al.,* 2010).



Integer value datasets

	ChIP-seq	RNA-seq	FAIRE	Other
bedgraph	845,755,171	108,953,786	4,915,956,397	1,141,962,427
gzip_bg	189,587,753	26,150,036	1,102,539,306	257,052,301
bigwig	208,576,900	28,471,960	1,743,598,761	339,320,618
val_delta	81,975,438	11,373,861	434,738,525	100,079,369
diff_delta	80,339,245	9,814,877	372,053,219	94,906,053
huff128	74,211,413	9,208,952	315,401,311	86,642,336
huff1024	74,211,738	9,242,983	315,401,487	86,725,800

Floating-point-value datasets

	ChIP-seq	RNA-seq	FAIRE	Other
bedgraph	781,334,516	400,658,833	8,638,293,826	7,909,472,427
gzip_bg	179,346,710	90,061,435	1,906,805,910	1,782,078,373
bigwig	207,948,348	88,773,468	3,013,151,266	1,622,209,234
val_delta	84,136,989	51,425,775	767,467,633	701,190,732
diff_delta	79,366,666	42,389,237	587,010,551	494,293,910
huff128	72,172,695	37,806,248	499,523,704	434,732,823
huff1024	72,175,662	37,202,813	499,579,990	435,213,395

Fig. 6. This table indicates the mean file sizes for storing ChIP-seq, RNA-seq and Other data types using the raw text format (bedGraph) and six different compression schemes. The bars in the background show the relative ratios between the compression schemes

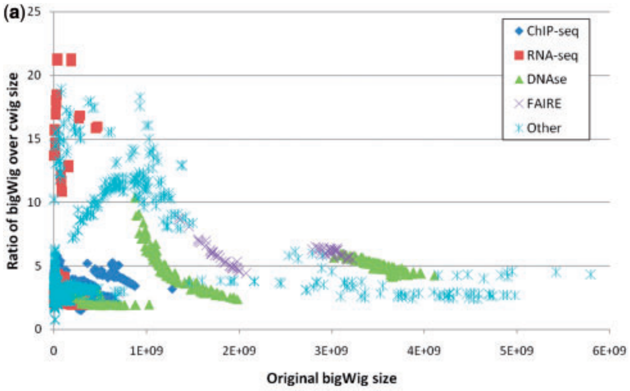
5.1 File sizes comparison

Compare different methods: Figure 6 shows results that compare different storage formats for different data types. The methods used in this experiment are (i) bedGraph is the raw text format of the input file, (ii) gzip_bg is the gzip compressed bedGraph format, (iii) bigWig is the method from UCSC, (iv) val_delta is our method that stores the raw signal values using delta code, (v) diff_delta is our method that stores signals by their differences using delta code only and (vi) huff128 and (vii) huff1024 are our methods that store signals by their differences using a mix of Huffman code and delta code. huff128 encodes the most frequent 127 values by unique Huffman symbols, whereas the rest of the values are encoded by delta code. huff1024 is similar to huff128; but the number of Huffman symbols are 1023.

For clarity, Figure 6 shows only four types of data: ChIP-seq, RNA-seq FAIRE and Other. (For full result, please refer to Supplementary B.) The bars in the background show the relative ratios between the compression schemes. Among our methods, huff128 and huff1024 are consistently better than val-delta and diff-delta. huff128 and huff1024 give similar size. This supports the observations in Section 3.2 that, higher number of Huffman symbols does not improve compression. Based on this experiment, we choose huff128 as our default compression method for cWig format.

Compared with bigWig and gzip, our methods use at most half of their sizes. In most of the files, the file sizes of bigWig and gzip are similar because the bigWig uses gzip to compress their main data. However, for high-resolution files, e.g. FAIRE data type, bigWig uses considerably more space than gzip. We found that this space is usually accounted for its indexing structures to support random access and queries.

Compare ours and bigWig: Figure 7 compares the file sizes between cWig and bigWig formats. Figure 7a plots the original



(b)

	Integer	Float	Both
ChIP-seq	2.88	3.27	3.09
RNA-seq	3.50	2.85	3.35
DNAse	3.04	5.10	3.43
FAIRE	5.77	6.05	5.91
Other	5.62	2.55	4.73
All_UCSC	**3.89**	**3.27**	**3.65**

Fig. 7. Ratio between bigWig file sizes and our file sizes. (**a**) The compression ratio for all UCSC files whose sizes are <6 GB. (The files that are excluded are three FAIRE files and the liver cancer file). (**b**) The mean of the compression ratio between bigWig and our format for each file type. (The last column represents both integer-value and floating-point-value files. 'All UCSC' row represents all the files types in UCSC. The light scale of the cells of the table is in proportion to the value inside)

bigWig size versus the reduction that we can achieve. Figure 7b is a table that summarizes the ratios based on the data types. It shows that our format is (in average) 3.6 times smaller than bigWig. In particular, cWig is more compressible for high resolution datasets, e.g. FAIRE and DNase.

We noticed some users truncate the significant digits of the values to reduce the file sizes of bigWig. We conducted an experiment to investigate its effect on both formats. The detail is included in the Supplementary D.

5.2 Running time comparison

Linear compression and extraction: Figure 8 shows the average compression/decompression speed for different methods. Because the compression/decompression speed is consistent with the input file size, we only show the average processing time in terms of megabytes per second. The figure shows that bigWig and gzip have similar compression speed. Our program is about two times faster. For decompression speed, our program is ~150% faster than bigWig, but slower than gzip.

Random queries: This set of experiments measure the query speed of operations coverage, minimum and average for both our tool and bigWig tool. We tested three sets of queries: (i) each query is a random interval. The order of queries is also random. (ii) Each query is a random interval. But the list of queries is arranged in increasing order of the start positions. (iii) The query intervals are the confirmed human gene regions. We call this set 'real queries' set. (It contains 76 969 intervals. This set is intended to simulate the actual list of queries made by the bioinformaticians).

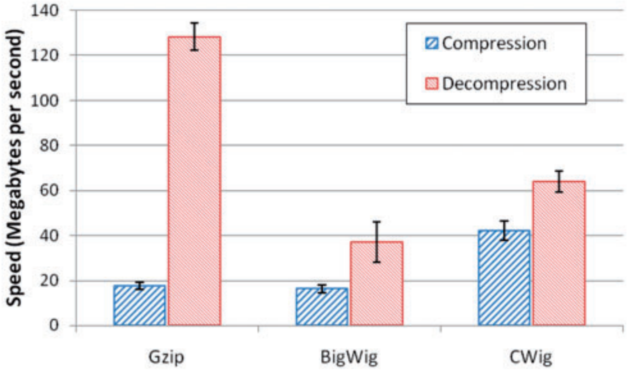


Fig. 8. Average compression and decompression speed in Megabytes per second (higher is better) with SDs for each method

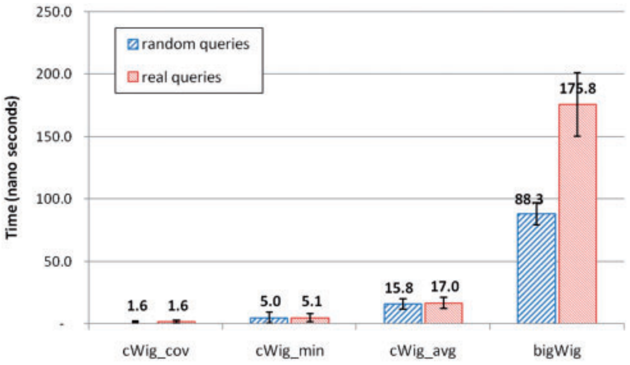


Fig. 9. Average query time in nanoseconds (lower is better) for randomly generated queries and gene region queries

Because the speed of both programs for query types (i) and (ii) are not significantly different, we only summarize the speed for query types (i) and (iii). In addition, because the query speed for the three operations in bigWig is similar, we only report the average query speed of bigWig.

Figure 9 shows that the query speed of our program is ~10–100 times faster than that of bigWig, depending on query type. In our program, coverage queries are much faster than the minimum and the average queries because coverage queries only use the interval position component. The minimum queries are faster than the average queries in sparse files where there are a lot of regions without values.

We noticed that there is a big difference in bigWig speed between random queries and real queries. After some investigations, we found that bigWig query speed may be affected by the query interval length. It is slower for shorter intervals. We create a query file that has the same starting positions as the real query file, but increased in the interval lengths. bigWig is much faster when the interval lengths are larger than 1 million bases. Note that the average interval length of the random query in Figure 9 is around half the chromosome length, whereas the average interval length of the genes is only 54 783.

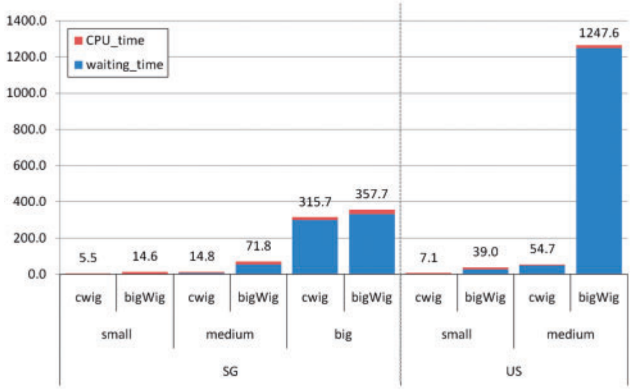


Fig. 10. cWig and bigWig remote query time (in seconds)

5.3 Remote file access speed

In this experiment, we measure the query speed in different network conditions. We select three input files of different sizes from UCSC hg19 database. They are called 'small', 'medium' and 'big'. The sizes of the corresponding bigWig files are 824 KB, 98 MB and 5.5 GB, respectively. The sizes of the corresponding cWig files are 414 KB, 35 MB and 1.4 GB, respectively.

The query speed is measured in two different network conditions: 'SG' and 'US'. 'SG': the files and the programs are both hosted in Singapore and connected through the Internet. The average round trip time is ~100 ms; the bandwidth is ~5–10 MB/s. 'US': the programs are in Singapore, and the files are hosted in California, USA. The round trip time is ~210 ms, the bandwidth is ~300–850 KB/s.

Similar to the previous experiment, we use the human genes regions as the query set.

Figure 10 compares the running times of bigWig and cWig under different network conditions and using different input files. (Note that, there is no measurement for big file under 'US' network condition owing to our resource limitation.) In these experiments, the CPU times of both programs are accounted for <10% of the total running times for medium and big input files. The programs spend most of their time waiting for network responses.

Our file size significantly helps in the experiments on the medium file. Because cWig file is smaller, the queries on this file get cached in fewer iterations. For small file, the time difference is not significant. Both programs can cache the small file after a few queries. For big file, both programs fail to cache the file, and hence, both methods spend similar amount of time to wait for the network to respond.

6 CONCLUSION

This article proposed the file format for cWig to store signal data. Comparing with bigWig, cWig not only uses lesser space but also provides faster queries. This format should be useful for visualization applications like UCSC genome browser (Karolchik *et al.*, 2014) and Broad Institute Integrative Genome Viewer (Robinson *et al.*, 2011) and for Biologists to analyze and discover features in their data. In the future, we would like to extend our idea to represent other types of data [like bigBed (Kent *et al.*, 2010) and BAM (Li *et al.*, 2009)]. We also want to consider lossy compression methods to gain better compression over noisy data.

Conflicts of Interest: none declared.

REFERENCES

Bock,C. (2012) Analysing and interpreting DNA methylation data. *Nat. Rev. Genet.*, **13**, 705–719.

Cock,P.J. *et al.* (2010) The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res.*, **38**, 1767–1771.

Cover,T. and Thomas,J. (1991) *Elements of Information Theory*. Wiley, New York, NY, USA, Chapter 7.6, p. 195.

Danecek,P. *et al.* (2011) The variant call format and VCF tools. *Bioinformatics*, **27**, 2156–2158.

Elias,P. (1975) Universal codeword sets and representations of the integers. *Inf. Theory IEEE Trans.*, **21**, 194–203.

Fritz,M.H.Y. *et al.* (2011) Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, **21**, 734–740.

Gundersen,S. *et al.* (2011) Identifying elemental genomic track types and representing them uniformly. *BMC Bioinformatics*, **12**, 494.

Guttman,A. (1984) R-trees: a dynamic index structure for spatial searching. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD'84. ACM, New York, NY, pp. 47–57.

Hoffman,M.M. *et al.* (2010) The genomedata format for storing large-scale functional genomics data. *Bioinformatics*, **26**, 1458–1459.

Hu,Y. *et al.* (2013) DiffSplice: the genome-wide detection of differential splicing events with RNA-seq. *Nucleic Acids Res.*, **41**, e39.

Huffman,D. (1952) A method for the construction of minimum-redundancy codes. In: *Proceedings of the I.R.E.* Springer India, India, pp. 1098–1102.

Karolchik,D. *et al.* (2014) The UCSC genome browser database: 2014 update. *Nucleic Acids Res.*, **42**, D764–D770.

Kent,W.J. *et al.* (2010) BigWig and BigBed: enabling browsing of large distributed datasets. *Bioinformatics*, **26**, 2204–2207.

Li,H. *et al.* (2009) The sequence alignment/map (SAM) format and SAMtools. *Bioinformatics*, **25**, 2078–2079.

Liu,T. *et al.* (2011) Cistrome: an integrative platform for transcriptional regulation studies. *Genome Biol.*, **12**, R83.

Okanohara,D. and Sadakane,K. (2007) Practical entropy-compressed rank/select dictionary. In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, New York, NY, USA.

Patrascu,M. (2008) Succincter. In: *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. IEEE, New York, NY, USA, pp. 305–313.

Raman,R. *et al.* (2002) Succinct indexable dictionaries with applications to encoding k-Ary trees and multisets. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA'02. Society for Industrial and Applied Mathematics, New York, NY, USA, pp. 233–242.

Robinson,J.T. *et al.* (2011) Integrative genomics viewer. *Nat. Biotechnol.*, **29**, 24–26.