Journal of Big Data

METHODOLOGY Open Access

StreamAligner: a streaming based sequence aligner on Apache Spark

Sanjay Rathee[1*†] and Arti Kashyap[1,2†]

*Correspondence: sanjaysinghrathi@gmail.com
†Sanjay Rathee and Arti Kashyap contributed equally to this work.
[1] School of Computing and Electrical Engineering, IIT Mandi, Kamand Campus, Mandi, India
Full list of author information is available at the end of the article

Abstract
Next-Generation Sequencing technologies are generating a huge amount of genetic data that need to be mapped and analyzed. Single machine sequence alignment tools are becoming incapable or inefficient in keeping track of the same. Therefore, distributed computing platforms based on MapReduce paradigm, which uses thousands of commodity machines to process and analyze huge datasets, are emerging as the best solution for growing genomics data. A lot of MapReduce-based sequence alignment tools like CloudBurst, CloudAligner, Halvade, and SparkBWA are proposed by various researchers in recent few years. These sequence aligners are very fast and efficient. These sequence aligners are capable of aligning billions of reads (stored as fasta or fastq files) on reference genome in few minutes. In the current era of fastly growing technology, analyzing huge genome data fast is not enough. We need to analyze data in real time to automate alignment process. Therefore, we propose a MapReduce-based sequence alignment tool StreamAligner which is implemented on Spark streaming engine. StreamAligner can align stream of reads on reference genome in real time. Therefore, it can be used to automate sequencing and alignment process. It uses suffix array index for read alignment which is generated using distributed index generation algorithm. Due to distributed index generation algorithm, index generation time is very less. It needs to upload index only once when StreamAligner is launched. After that index stays in Spark memory and can be used for an unlimited times without reloading. Whereas, current state-of-the-art sequence aligner either generate (hash index based) or load (sorted index based) index for every task. Hence, StreamAligner reduces time to generate or load index for every task. A working and tested implementation of streamAligner is available on GitHub for download and use. We tested the effectiveness, efficiency, and scalability of our aligner for various standard and real-life datasets.

Keywords: Sequence alignment, Apache Spark, Hadoop, Distributed computing frameworks

Introduction

The trend of using latest computer technology to manage biological information is on the rapid rise during last decade. Currently, computers are used to collect, store, manage, analyze and integrate genetic and biological information. Therefore, bioinformatics has emerged as a very popular research area in last decade. Sequence alignment is like the heart of bioinformatics field and has attracted huge attention by researchers. Sequence alignment is a way to identify regions of similarity between two sequences of genome data. Sequence alignment has various applications like identifying homologous proteins, analyzing gene expressions and mapping variations between individuals. Sequence alignment helps bioinformaticians to understand genome and find the answers related to similarity and dissimilarity between two genomes. Recently, Riccardo Sabatini [1] in his TEDx talk, showed how they are able to read the genome and build a human from this information. They find out the sequences which are responsible for dissimilarity between humans and used this information to make a human face from his/her DNA.

A large number of sequence aligners have been proposed by researchers during last decade. Most of these sequence aligners were either hash based or sorted index based.

Hashing based aligners use hash trees or hash tables to store hash values of either the query genome or the reference genome and then use the un-hashed genome as a single probe against the hash table. Techniques like MAQ [2], Eland [3], SeqMap [4], ZOOM [5] and RMAP [6] use hashing techniques to hash the read sequence and scan through the reference genome. These techniques have the drawback of having to scan the entire genome even when very few reads need alignment, which results in longer computation times. Tools like NovoAlign [7], SOAPv1 [8], PASS [9], MOM [10], BFAST [11] and ProbeMatch [12] also employ hashing techniques to hash the genome. While these methods can easily be parallelized, they require a lot of memory to store the index built for the reference genome.

To bypass the large memory requirement, *slider* [13] proposes a sequence alignment by merge-sorting the reference genome subsequences and read sequences. Recently, string matching algorithms based on the *Burrow-Wheeler Transformation (BWT)* [14], which is a string compression technique, has drawn the attention of many research groups. Techniques like Bowtie [15], BWA [16] and BWA-SW [17] which are based on BWT [14], have also become very popular due to their vastly improved memory efficiency and their support for flexible seed lengths. These BWT-based sequence alignment tools provide fast mapping of short reads of DNA sequences against a reference genome sequence with small memory footprint using a data structure like FM-Index [18] built atop the BWT. These studies use sorting algorithms for matching. Therefore, they are highly accurate with accuracies as high as 99.9%.

Most of these sequence alignment tools were very efficient and accurate until the introduction of *Next-Generation Sequencing (NGS)*. NGS has led to a huge amount of sequencing data which has rendered many existing sequence alignment tools obsolete. For example, NGS technologies like *Illumina HiSeqX* can easily generate nearly 6 billion sequence reads per run. After sequencing, mapping these read sequences onto a reference genome is the most important task in a sequence analysis work-flow. Alignment of such large volumes of read data onto a reference genome is a very time consuming task. Many state-of-the-art sequence aligners are developed to handle this huge amount of data efficiently but NGS platforms are evolving so rapidly that they push sequencing capacity to unprecedented levels. Therefore, sequence alignment still remains a bottleneck for bioinformaticians.

To combat this deluge of NGS data, some sequence aligners based on big data technologies have been proposed in the last few years. Big data technologies like *Hadoop* [19] and *Spark* [20] based on the MapReduce paradigm use distributed computing to handle massive volumes of data very effectively and efficiently. Early initiatives towards the trend of using MapReduce [21] based platforms like Hadoop and Spark for sequence alignment have already been taken, such as CloudBurst [22], CloudAligner [23], BlastReduce [24], BigBWA [25] and SparkBWA [26] to name a few. The results are very effective and promising.

Sequence alignment tools with distributed computing capabilities like CloudBurst and CloudAligner are faster and more efficient with short reads but they have lower accuracy and are unable to handle long reads. Sequence alignment tools like BigBWA [25], Halvade [27] and SparkBWA [26] are very accurate but they suffer from high time/space complexity for index generation.

Current state-of-the-art sequence aligners are very accurate and efficient. They are capable to handle large genomes very efficiently. They can align billions of reads in few minutes. But, future technologies are converging towards automated and real-time processing tools. Nearly all state-of-the-art sequence aligners start alignment process when whole read data is available. They can not process stream of reads. Therefore, we propose a new sequence aligner which can align stream of reads on reference genome in real time. It will help in making DNA matching process automated where a read can be aligned as soon as it is produced by NGS machines. It uses a partitioned suffix array index to align reads on reference genome.

StreamAligner has two main advantages over state-of-the-art sequence aligners. Firstly, it is the only sequence alignment tool which can align stream of reads on reference genome to give real time results. Secondly, it needs to upload sorted suffix index only once for unlimited tasks whereas all state-of-the-art sequence aligners either need to generate (hash based aligners) or upload (sorted index based aligners) index for every task.

In this paper, we discuss capabilities of Spark Streaming engine and some important related works. Then, we give a detailed description of our MapReduce based tool StreamAligner and its API. Thereafter, we focus on the evaluation of StreamAligner for different datasets on the available cluster. At last, we talk about conclusions and future directions.

Mapreduce and Spark streaming

With the availability of cloud computing technologies like Amazon EC2 cloud and Microsoft Azure, as evolutions to accommodate computing and storage service as a utility at very affordable prices, users can use these cloud services via Internet from home or workplace by paying for resources consumed without worrying about availability, maintenance and flexibility issues. Cost is based on the type of service and time of service. Therefore, cloud computing has emerged as a very promising solution for demands of storage and computation in bioinformatics eliminating the need of powerful and high computing capacity server. But to achieve efficiency, performance, and scalability for processing huge data, a highly parallel distributed computing model is required. Therefore, a parallel computing framework called MapReduce [21] was designed by Google which allows using thousands of commodity machines in parallel. MapReduce framework works on a basic idea of the flow of $\langle key, value\rangle$ pairs through map and reduce phases. Input is split into fixed size chunks and distributed over available mappers. Every mapper processes its chunk of data and generates $\langle key, value\rangle$ pairs. These $\langle key, value\rangle$ pairs are shuffled or sorted to group values based on keys to generate intermediate

$\langle key, value \rangle$ pairs where all values with same key are grouped together. Reducers take intermediate $\langle key, value \rangle$ pairs and combine all values for a key to generate final results. MapReduce framework can handle huge datasets because all map and reduce operations are executed concurrently on many machines. All map tasks need to finish to start reduce tasks. Many MapReduce framework based tools like Hadoop, Spark, and Flink are available to analyze huge datasets with ease for different applications [28]. Hadoop is widely used MapReduce based model in bioinformatics in recent few years [29]. Sequence alignment tools like CloudBurst [22], CloudAligner [23] and BlastReduce [24] used Hadoop for heavy analysis. Though Hadoop provides a highly parallel computing environment but has a limitation of high I/O time during various iterations. Apache Spark [20] overcomes this limitation of Hadoop by using its in-memory computing technique with help of RDD storage. I/O operations on Spark RDD [20] are very efficient and fast due to which sometimes it outperforms Hadoop by 100 times [30]. These advantages of Apache Spark ignited an interest in us to use Apache Spark for sequence alignment tool StreamAligner. In recent few years, technologies which can process data in real-time has gained a lot of attention [31, 32]. Therefore, we implemented StreamAligner in such a way so that it can process real-time stream data. Spark streaming API strengths our aligner because it enables scalable, high-throughput, fault-tolerant stream processing of live data streams. It can take stream of data from many sources like HDFS, Kafka, Flume, Kinesis, or TCP sockets. It uses distributed computing to process this stream data and can store results on filesystem, databases or dashboard.

Related work

A large number of sequence alignment tools can be found in literature which are very efficient and powerful in mapping reads to the reference genome. With the evolution of NGS machines, the size of reads data has increased so fast that single machine based sequence aligners were unable to keep track with same. Therefore, sequence alignment tools based on parallel and distributed computing architecture evolved as the best option for bioinformaticians. MapReduce [21] based platforms like Apache Hadoop [19] and Spark [20] has gained a lot of attention in recent few years as distributed computing based platforms. Many sequence aligners which use big data technologies like Apache Hadoop and Spark were implemented in last few years. CloudBurst [22], CloudAligner [23], Halvade [27], SEAL [33], BigBWA [25] and SparkBWA [26] are mostly used sequence aligners which use big data technologies.

CloudBurst is a read-mapping algorithm modeled after RMAP [6] and implemented on Hadoop. It uses MapReduce [21] framework to run a task on many machines parallelly. Tools like CloudBurst, SeqMapReduce [34] and Crossbow [35] which use seed-and-extend approach and implemented on Hadoop were very promising but they have many limitations. CloudBurst does not support the fastq format which is most common output file format for current NGS machines. It also uses only command line interface which is not user-friendly. Website and code for sequence aligners like SeqMapReduce are inaccessible. Crossbow has a very user-friendly interface and uses Bowtie [15] and Soap-snp tools with perl and shell scripts on Amazon EC2 cloud [36]. But it inherits limitation like only three mismatches allowed in bowtie and map only short reads. All these limitations were tackled by a new sequence alignment tool CloudAligner. It has better performance, user-friendly interface, and support for long reads. CloudAligner is very promising sequence aligner still it is not so efficient and accurate as Burrow-Wheeler based sequence aligners which use distributed computing based architectures to enhance BWA [16] performance.

SEAL is one of the best MapReduce based sequence aligners which is implemented in Python and runs on Hadoop platform. It uses Python to write program and wrapper to call BWA. It has some limitations like it works with only a few modified version of BWA and doesnot support BWA-MEM [17] algorithm for long reads.

Francesco Versaci came with a tool which takes raw data (BCL files) as input and produces aligned DNA sequences [37]. It uses a Flink based tool to convert BCL format into fastq format. Then, SEAL API align reads to the reference genome. It suffers from the limitation that it still uses Hadoop based sequence aligner whereas many faster aligners based on more promising and faster distributed computing platform like Spark exist.

Halvade also works on top of Hadoop platform. It has some limitations like mappers needs to call BWA as an external process which can cause timeout during execution of Hadoop. Therefore, task timeout parameters need to be configured adequately which need a priori knowledge about application execution time. Later, many parallel programming based aligners like pBWA [38] which uses MPI to parallelize BWA, were designed. Lack of fault tolerance and no support for BWA-MEM [17] were the main drawback of pBWA.

Many sequence aligners used the power of GPUs to enhance the performance of BWA. BarraCUDA [39], as suggest by name works on CUDA programming model. It supports BWA version 0.5.x. and still, have the limitation that it supports only BWA-backtrack algorithms for short reads.

Most of these tools used Hadoop for distributed computing which has the limitation of high I/O time during iteration. Therefore, a highly distributed computing based platform Apache Spark which outperformed Hadoop by a huge margin for various machine learning problems has got a lot of attention these days. Already many Spark based sequence aligners are proposed by researchers. MetaSpark is one of these sequence aligners which is implemented on Spark [40]. It aligns metagenomic reads on reference genome quite fastly. It constructs k-mers for reference genome and reads and save it in Spark RDD. Seeding, filtering and banded alignment process on reference seeds and read k-mers produce final alignment results. It has a limitation that it will become inefficient for large datasets because Spark need to store k-mers in memory (RAM) and memory usage for storing k-mers is too high.

Recently, a Spark based sequence aligner called SparkBWA has evolved as most efficient and promising sequence aligner. SparkBWA has shown great performance and outperformed nearly all existing sequence alignment algorithms. SparkBWA calls BWA [16] algorithms using Java Native Interface (JNI)and supports all versions of BWA. SparkBWA still has some limitation like no-support for stream processing and high index generation time. Our sequence alignment tool StreamAligner resolved these limitations very efficiently.

Methods

StreamAligner is a MapReduce based sequence alignment tool implemented on Apache Spark. It uses a suffix array index to map reads onto a reference genome. It uses three iterations to build an index and map a read onto a reference genome.

Reference preprocessing

In the first iteration, we clean and transform the reference genome so that it can be processed using distributed computing. Most of the state-of-the-art index based sequence aligners like SparkBWA, BigBWA, and Halvade use a BWA tool to generate their index using a sequential approach. Therefore, index generation time for BWA is higher on a large cluster too. To combat this aforementioned issue, StreamAligner uses a distributed algorithm to generate a suffix array index. Therefore, StreamAligner must get data which can be processed independently.

Index generation

In the second iteration, we generate a suffix array index for cleaned and transformed reference genome. StreamAligner uses a distributed algorithm to generate a suffix array index. Therefore, StreamAligner outperforms nearly all existing state-of-the-art indexing tools like BWA in terms of index build time. Additionally, the StreamAligner index generation time decreases linearly with increase in the size of the cluster. Suffix array indexes generated in our first iteration are stored in different partitions depending on the prefix of our suffixes, i.e., indexes for suffixes starting with *a*, *c*, *g*, *n* and *t* will be mapped to different partitions.

Read mapping

In the third iteration, we map streams onto the reference genome. SparkBWA, BigBWA, and Halvade use the entire genome BWT index to map reads onto the reference genome. In contrast, StreamAligner uses only a single partition of the index based on the first character of the read to map reads onto the reference genome. With a higher number of partitions, the mapping computations become much fewer in number which results in quicker mapping.

Phase I-Reference preprocessing

Initially, the reference genome which is stored on HDFS[1] [19], is given as input to the mappers. The reference genome file is chunked and distributed across the mappers, where each mapper receives a set of lines. The detailed process of generating a suffix-array for the reference genome is outlined in Algorithm 1.

[1] Hadoop Distributed File System.

Algorithm 1 Phase I: Step I- Transformation and cleaning of reference genome

```
Input: Reference Genome $R_g$
Output: Transformed and cleaned Reference Genome $R_t$
 1: procedure TRANS-REFERENCE–GEN
 2:     for each line $l_i \in R_g$ do
 3:         flatMap($lineoffset, l_i$).zipWithIndex
 4:         Yield($index, lineoffset + l_i$)
 5:         end flatMap
 6:         storeAtRDD1
 7:     for each line $l_j \in R_g$ do
 8:         flatMap($lineoffset, l_j$).zipWithIndex
 9:         Yield($index - 1, l_j$)
10:         end flatMap
11:         storeAtRDD2
12:     RDD3=RDD1.joinByKey(RDD2).values
13:     $R_t$=RDD3.saveAsTextFile
```

We use the *Hadoop configuration function* so that every line in the file has an index associated with it, where this index represents the *location of the first character of the line in the reference genome* (lines 2). Subsequently, a *sortByKey* function on reference genome *JavaRDD* generates $\langle key, value\rangle$ pairs where *value* denotes concatenation string of location of first character of line and line itself and *key* is the number for line in reference genome starting from 1 for first line (lines 3–5). Output $\langle key, value\rangle$ pairs stored at $JavaRDD_1$ (line 6). Parallely, a *sortByKey* function on reference genome *JavaRDD* generates $\langle key, value\rangle$ pairs where *value* denotes the line and *key* is the number for line in reference genome starting from 0 for first line (lines 8–11). Output $\langle key, value\rangle$ pairs stored at $JavaRDD_2$ (line 12). At last, a *joinByKey* function joins $JavaRDD_1$ and $JavaRDD_2$ to produce $\langle key, value\rangle$ pairs where *key* denotes line number and *value* is the concatenation string of values from $JavaRDD_1$ and $JavaRDD_2$ (line 14). All the values from $\langle key, value\rangle$ pairs will be stored on HDFS as a text file (line 15). The *lineage graph* in Fig. 1 shows the functional flow of Algorithm 1.

Figure 2 shows an example of the reference preprocessing step. A reference genome stored on HDFS is treated as the input. The mappers in the first round receive a set of lines (e.g. mapper MAP-1 receives lines L1–L3) and apply *zipWithIndex* function on each line and generate corresponding $\langle key, value\rangle$ pairs. For example, Line L1 (*ACG TTCA*) is mapped to $\langle 1, 1 + ACGTTCA\rangle$. Output $\langle key, value\rangle$ pairs stored at $JavaRDD_1$. The mappers in the second round receive a set of lines (e.g. mapper MAP-1 receives lines L1–L3) and apply *zipWithIndex* function on each line and generate corresponding $\langle key, value\rangle$ pairs. For example, Line L1 (*ACGTTCA*) is mapped to $\langle 0, ACGTTCA\rangle$. Output $\langle key, value\rangle$ pairs stored at $JavaRDD_2$. Subsequently, a *joinByKey* function joins $JavaRDD_1$ and $JavaRDD_2$ to produce corresponding $\langle key, value\rangle$ pairs. For

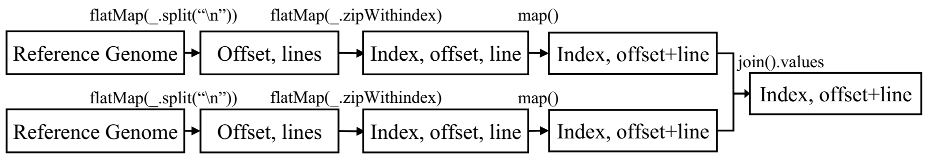

Fig. 1 Lineage graph for Phase I (reference preprocessing)

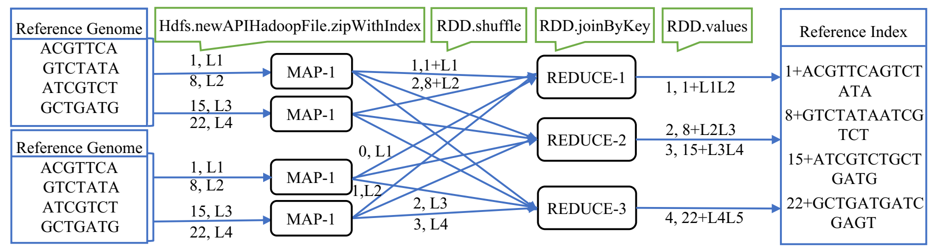

Fig. 2 Mapreduce architecture for Phase I (reference preprocessing)

example, tuple $\langle 1, 1 + ACGTTCA\rangle$ from $JavaRDD_1$ will join with tuple $\langle 1, GTCTATA\rangle$ in *JavaRDD*2. Output will be stored on HDFS as text file.

Now, to estimate the time complexity, let us take, n as the total number of base pairs in reference genome and p as the number of base pairs in every line of reference genome. Then algorithm takes total n/p cycles to generate *RDD*1. For a cluster with m number of mappers (cores), $mapper_1$ will take n/mp cycles. The time complexity for $mapper_2$ will also be same as $mapper_1$. For a cluster with r number of reducers, join function will take n / rp cycles. Hence, total complexity for reference preprocessing will be $(2nr + mn)/(mrp)$.

Phase II-Index generation

Preprocessed reference genome will be chunked and distributed across mappers, where each mapper receives a set of lines. The detailed process of generating a suffix-array for the reference genome is outlined in Algorithm 2.

Algorithm 2 Phase I- Suffix Array Index Generation

```
Input: Transformed and cleaned Reference Genome $R_t$, Keylength $\omega$
Output: Index of Suffix ($\alpha$), Suffix Array for A $S_a$, Suffix Array for C $S_c$, Suffix Array for G $S_g$, Suffix
    Array for T $S_t$
 1: procedure SUFFIX−GEN
 2:     for each line $t \in R_t$ do
 3:         flatMap($lineoffset + l_i + l_j$)
 4:         for each char $c \in l_i$ do
 5:             $\alpha$=$lineoffset$+location of $c$ in $l_i$
 6:             $s$ = substring($\alpha$, $\alpha$ + keylength)
 7:             Yield($s$, $\alpha$)
 8:         end flatMap
 9:         storeAtRDD1
10:     $S_a$=RDD1.filter("a").sortByKey.values()
11:     $S_c$=RDD1.filter("c").sortByKey.values()
12:     $S_g$=RDD1.filter("g").sortByKey.values()
13:     $S_t$=RDD1.filter("t").sortByKey.values()
```

We use simple *readTextFile* for loading preprocessed reference genome on *JavaRDD* of Spark. A *flatMap* task on this *JavaRDD* will yield $\langle key_n, value_n\rangle$ pairs where key_n is a string of characters of length k starting from the position in the reference genome given by the $value_n$ field (lines 2–9). Now, these keys are sorted in the shuffle task (*sortByKey*) and reducers combine all the outputs of the shuffle task and partition the final sorted results according to the prefix of the suffixes (lines 12–15). The *lineage graph* in Fig. 3 shows the functional flow of Algorithm 2.

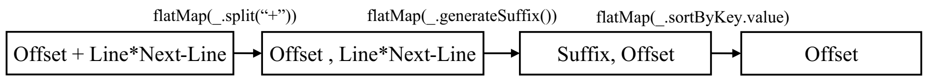

Fig. 3 Lineage graph for Phase II (suffix index generation)

Figure 4 shows an example of the index generation step. A preprocessed reference genome stored on HDFS is treated as the input. The mappers in the first round receive a set of lines (e.g. mapper MAP-2 receives lines L2–L3) and apply *flatMap* task on each line and generate corresponding $\langle key, value \rangle$ pairs like $\langle ACG, 1 \rangle$ for every line where ACG denotes a suffix of fixed size and 1 is the starting location of that suffix in the reference genome. Thus, for line L1 ($ACGTTCA$), the mapper produces $\langle ACG, 1 \rangle$, $\langle CGT, 2 \rangle$, $\langle GTT, 3 \rangle$, $\langle TTC, 4 \rangle$, $\langle TCA, 5 \rangle$, $\langle CAG, 6 \rangle$, and $\langle AGT, 7 \rangle$. These keys are then sorted in the shuffle task (*sortByKey*) and the reducers combine all the outputs of the shuffle task and partition the final sorted results according to the prefix (a, c, g, n, t) of the suffixes. For line L1, the reducer outputs $\langle ACG, 1 \rangle$, $\langle AGT, 7 \rangle$.

In this phase, mappers will take preprocessed reference having n/p lines as input. For every line of preprocessed reference genome, map task will take $3p$ cycles to generate suffixes. With a cluster of m mappers, it will take $3n/m$ ($3p * (n/p) * (1/m)$) cycles to generate all suffixes. Now, a filter and *sortByKey* function will add $(n + np)/m$ cycles to make total time complexity to be $(4n + np)/m$.

Phase III-Read mapping

The second iteration of AVLR-Mapper finds the location of reads in the reference genome by using the suffix array index of that reference genome. Algorithm 3 describes the process of mapping stream of reads on the reference genome. Initially, a stream of reads from sources like Kafka, HDFS or flume are distributed across mappers and each read is partitioned into *seeds* (Algorithm 3, lines 2–6). Let r and s denote a read and a seed, respectively. Additionally, let $\mathcal{L} : \mathbb{Z} \rightarrow \mathbb{Z}$ represent a length function. Then, $\mathcal{L}(s) = \mathcal{L}(r)/(e + 1)$, where e is the number of allowed errors.

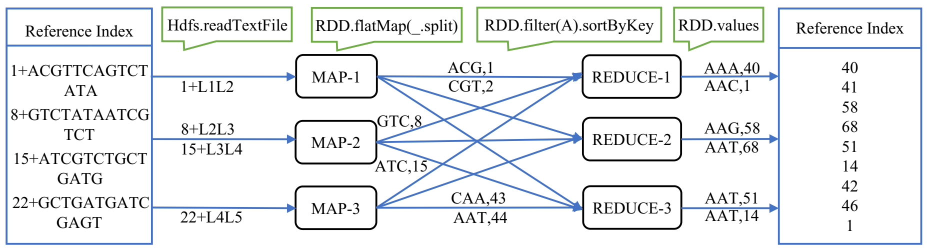

Fig. 4 Mapreduce architecture for Phase II (suffix index generation)

Algorithm 3 Phase II- Read Mapping

```
Input: Reads data ($R_d$), Number of errors allowed ($\nu$), Suffix array ($S_a$, $S_c$, $S_g$, $S_t$)
Output: Read location in $R_g(\mu)$
1: procedure READ–MAP
2:     for each read $r \in R_d$ do
3:         flatMap(line offset, $r$)
4:         $\Theta$= $r$.size()
5:         $r_s(i)$=seed $i$ of read $r$
6:         $r_s(i)$=$r$.substring($(i-1)$*$\theta/(\nu+1)$)
7:         for each seed $r_s(i) \in r$ do
8:             $\mu$= Find($r_s(i)$, $R_g$, $f$, $l$)
9:             if ($\mu > 0$) then
10:                extend $r_s(i)$ to $r$
11:                $\rho$ = mismatch($r$, $R_g(\mu)$)
12:                if ($\rho <= \nu$) then
13:                    $r$ found at $\mu$
14:                else
15:                    $d_e$ = edit-distance($r$, $R_g(\mu)$)
16:                    if ($d_e < \nu$) then
17:                        $r$ found at $\mu$
18:        Yield($r$, $\mu$)
19:        end flatMap
```

For every seed *s*, the mapper finds its corresponding match in the reference genome by using the suffix array index (Algorithm 4) and generates a ⟨*key*, *value*⟩ pair, where *key* is the seed *s* and *value* is the location of the seed *s* in the reference genome (Algorithm 3, lines 7–8). If an exact match of a seed *s* exists, then we extend it to the whole read and find mismatches for the whole read (Algorithm 3, lines 9–11). If mismatches are less than *e* then a read with its location in the reference genome is yielded (Algorithm 3, lines 12–13). If the mismatches are more than *e* then the edit distance for the whole read is calculated and if this computed edit distance is less than *e*, then a read with its location in the reference genome is outputted (Algorithm 3, lines 14–17). Finally, reducers combine all intermediate results and output every read with its location in the reference genome. *Lineage graph* (Fig. 5) for phase III shows the flow of reads during Phase III.

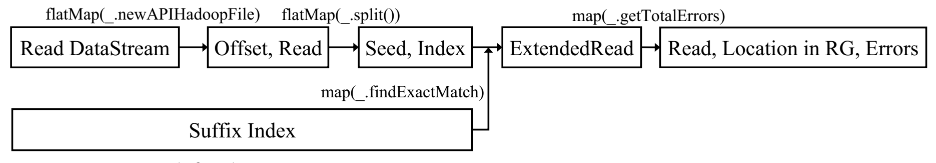

Fig. 5 Lineage graph for Phase III (stream mapping)

Algorithm 4 Find $(r_s,\ R_g,\ S_a,\ f,\ l)$

```
Input: Seed ($r_s$), Reference Genome ($R_g$), Suffix array ($S_a$), first of suffix array ($f$), last of suffix array ($l$)
Output: Location of $r_s$ in $R_g$
 1: procedure FIND-SEED
 2:     f=$f$
 3:     l=$l$
 4:     $\lambda$= length of $r_s$
 5:     $m$= (f+l)/2
 6:     $\tau$= $S_a(m)$
 7:     $\Phi$= $R_g(\tau, \tau+\lambda)$
 8:     if ($r_s$==$\Phi$) then
 9:         return $\tau$
10:     else if ($r_s$ <= $\Phi$) then
11:         Find($r_s$, $R_g$, $S_a$, f, $m$)
12:     else if ($r_s$ >= $\Phi$) then
13:         Find($r_s$, $R_g$, $S_a$, $m$, l)
```

Figure 6 shows a MapReduce architecture for an exemplary read data alignment on the reference genome using a partitioned suffix-array index. Initially, a stream of reads from data sources like Kafka, HDFS or flume are taken as input and stored as *JavaRDD*

A *flatMap* function on this *JavaRDD* distributes reads to different mappers. For example, mapper MAP-1 takes as input reads R1–R3. Reads are split into many seeds depending on the number of errors allowed. For example, read R1 (*ACGTCC*) is split into two seeds (*ACG* and *TCC*), if the number of permitted errors is 1. Exact matches for each seed is then sought in a suffix array partition depending on the start character of the seed. For example, if a seed begins with token *t*, then the exact match for this seed is searched only in $Index_t$. The search for exact matches for $Seed_1$ (*ACG*) and $Seed_2$ (*TCC*) is now restricted to only indices $index_a$ and $index_t$, respectively. If an exact match exists for the seed, then the seed is extended to the entire read. Continuing with our example, $seed_1$ (*ACG*) has an exact match at location 1–3 in the reference genome and hence it gets extended to the entire read, i.e., *ACGTCC*. Now, the *edit distance* between the *extended read* and reference location 1–6 is computed. A match is successful and is emitted when it is less than our allowed error threshold. For example, $read_1$ yields $\langle read_1, 1, 7\rangle$. Results for all reads are collected and stored in a text file on HDFS.

Finding the exact location of a seed in a reference genome is the most time-consuming task. But due to the indexed genome (suffix array), a maximum of 32 searches ($\log_2(3 \times 10^9)$) are required to search a seed in a reference genome, like a human genome, that contains 3 billion characters. We further reduce the cost of finding exact matches for seeds by partitioning our index. Our partitioning scheme requires that

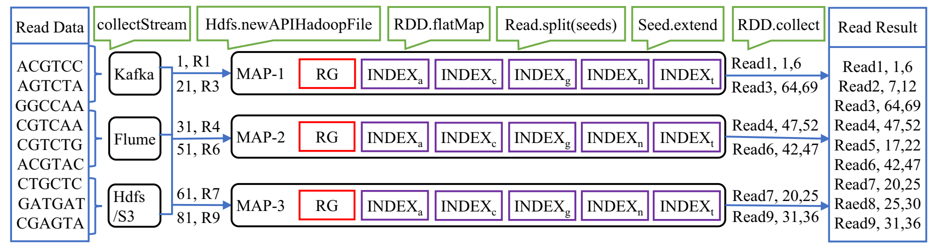

Fig. 6 Mapreduce architecture for Phase III (stream mapping)

suffixes starting with similar characters are placed in the same sorted partition. For example, if we have 5 partitions, then every read will be searched in nearly 1 / 5-th of the index, which reduces the number of searches for every read from 32 to nearly 30 or fewer. If the length of a read is 1000 bp then partitioning the suffix array saves nearly 2000 computations for every read. As the volume of reads is also massive (i.e., billions of reads), the time savings in computation are also quite significant. Let us assume that R is the size of a reference genome, q is the number of reads in a query sequence, n is length of a read, and the number of partitions p of our index is set to 5 (with prefix a, c, g, t, n). Then, the number of searches required to find a read in the whole genome is $\log_2 R$. Therefore, the number of searches required to find a read in a single partition is $\log_2 \frac{R}{p}$ and the number of searches reduced in a single partition is $\log_2 p$. As the total number of comparisons to search a read is n, the reduction in a total number of computations amounts to $nq \log_2 p$.

StreamAligner Api

One of the main targets of StreamAligner is to provide bioinformaticians an easy, simple and powerful way to perform sequence alignments using distributed computing based big data technology like Apache Spark. To achieve this goal, a basic API for StreamAligner is provided. StreamAligner can be started from Spark console. In Spark console, a Spark-submit command with necessary arguments is used to run StreamAligner. An example of a Spark-submit command with necessary arguments is shown in Fig. 7. All required commands to use StreamAligner are available in commands.txt file on GitHub directory. StreamAligner takes input as a stream of reads. These streams of reads can come from many sources like Kafka, HDFS/S3, and kinesis. The current version of StreamAligner is taking input from HDFS and storing results back to HDFS after processing. In the example given in readme.txt file on GitHub directory for StreamAligner, a sample query genome dataset is taken from HDFS as input. Results after processing are stored in a particular HDFS directory. Source code for StreamAligner is publicly available for users on GitHub directory so that users can edit the code to take streams from different sources according to their needs. We are flexible to get a request from users to add support from other stream sources in future.

```
/path/spark-submit
--class StreamAligner
--master spark://master:7077          # Connect to master of cluster
target/StreamAligner.jar              # StreamAligner Tool
hdfs://localhost:54310/chr1.fastq     # Reference genome
hdfs://localhost:54310/c-ref.txt      # Cleaned and transformed reference genome
50                                    # Number of base pair in single line of reference
20                                    # Keylength- size of keys to sort suffixes
hdfs://localhost:54310/refchr-index   # Chromosome with start location in reference
hdfs://localhost:54310/suffix-a       # Suffix array for suffixes starting with A
hdfs://localhost:54310/suffix-c       # Suffix array for suffixes starting with C
hdfs://localhost:54310/suffix-g       # Suffix array for suffixes starting with G
hdfs://localhost:54310/suffix-t       # Suffix array for suffixes starting with T
249250627                             # Total size of reference genome
hdfs://localhost:54310/query          # Query data stream source
"@HWI"                                # Query data delimiter
2                                     # Number of error allowed
hdfs://localhost:54310/result.fastq   # Location to save results on Hadoop
```

Fig. 7 Example running StreamAligner on Apache Spark from console

Results and discussion

We evaluated performance of StreamAligner for index generation and read mapping with different number of computing nodes and using several datasets of different sizes.

Cluster and dataset

We evaluated performance of StreamAligner on a cluster having five nodes where each node have 32 cores and 64 GB RAM. All computing nodes are running on Ubuntu 16.04.3 LTS operating system. *Oracle Java 8* is used to build project. *Spark 2.2.0* and *Hadoop 2.7* are used to run StreamAligner.

Mainly two types of datasets are required in sequence alignment applications. First one is reference genome and other one is query genome. Reference genome is a continuous string of millions of characters. Query genome contains reads of various length. We have used human *chromosome 1, chromosome 2, chromosome 3, chromosome 5, chromosome 21*, a full human genome (*hg19*), elephant genome (*loxAfr3*), rabbit genome (*oryKun2*) and cat genome (*felCat8*) as reference genomes for our experiments. All these reference genomes are sourced from the NCBI project [41]. Different subsets of *AML.fastq, BRL.fastq, GCAT, ERR000589, SRR015390, SRR062634, SRR642648* and *100k.fa* have been used as a query sequence. *100k.fa* was downloaded from the Cloudburst website, while *AML.fastq*, *ANL.fastq* and *BRL.fastq* are real-world datasets from *Agilent Technologies* [42]. The *ERR000589, SRR062634* and *SRR642648* dataset is from 1000genomes project website [43]. Detailed description of various datasets is given in Table 1.

Index generation

We evaluated StreamAligner's index generation tool in terms of scalability and efficiency.

The scalability of the index generation phase is evaluated by increasing the number of compute cores for Spark. Fig. 8 clearly shows that the index generation time decreases linearly with increasing compute cores.

Currently, *BWA* is one of the best index generation tools. Most of the recent sequence aligners like *BigBWA*, *SparkBWA*, and *Halvade* use *BWA* for index generation. We also compared the index generation times for *StreamAligner* and *BWA* for a fixed number of compute cores (64 cores) and different datasets. Figures 9 clearly shows that the index generation time of *StreamAligner* is nearly three times less than *BWA*.

Table 1 Query datasets description

Query genome	Number of reads	Bp per read	Size (MB)
100k.fa	100,000	36	4.18
BRL.fastq	3,958,076	100	1100
AML.fastq	304,745	150	112
ANL.fastq	2,986,312	400	2600
NA12750/ERR000589	$12 * 10^6$	51	3400
HG00096/SRR015390	$15.9 * 10^6$	51	5100
HG00096/SRR062634	$24.1 * 10^6$	100	11,800
150140/SRR642648	$98.8 * 10^6$	100	48,300

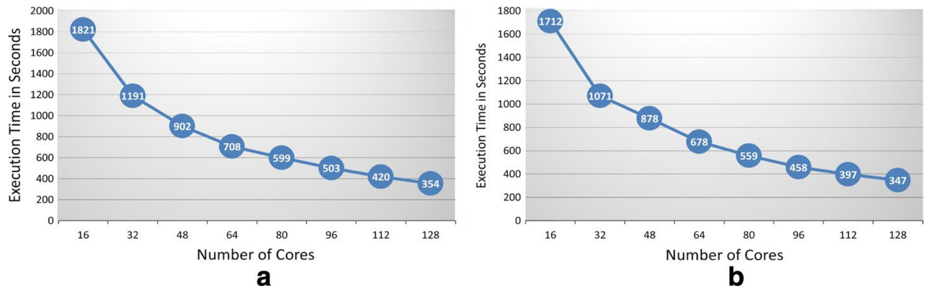

Fig. 8 Index generation time of (**a**) hg19 and (**b**) OryKun2 genome for StreamAligner

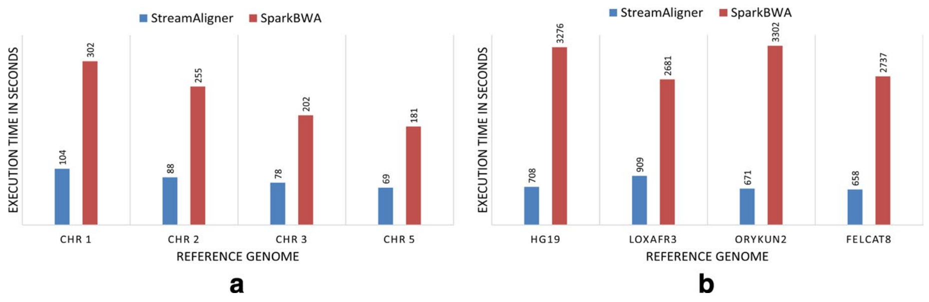

Fig. 9 Index generation time for StreamAligner and SparkBWA. **a** Performance for different chromosomes. **b** Performance for different genomes

Read mapping

StreamAligner is the only sequence aligner which can align stream of reads. Therefore, we can not show any comparative results for StreamAligner with any other sequence aligner for data streams. Still, we evaluated StreamAligner performance for static data (by feeding static read data as stream) and compared it with state-of-the-art sequence aligners. We evaluated StreamAligner in terms of performance, scalability and accuracy.

Execution time for mapping different read datasets on different reference genome is shown in Table 2. StreamAligner is capable to map millions of reads on large reference genome like human genome in few minutes. StreamAligner will perform better than SparkBWA when read length is high as we can see that speedup is high for ANL dataset. Results also show that performance of StreamAligner gets better in comparison to SparkBWA as the size of query dataset increases.

We evaluated scalability of read mapping phase by increasing number of computing nodes for Spark from 1 to 5. Figure 10 clearly shows that mapping time decreases as number of computing nodes increases. Therefore, we can say that StreamAligner is highly scalable and can align reads more efficiently as cluster size increases.

StreamAligner finds exact matches for non-overlapping seeds using binary search on sorted suffix array index. Therefore, it provides 100% accuracy for mapping reads with fixed number of allowed errors. Figures 11 and 12 show number of read matches found by StreamAligner, CloudBurst and SparkBWA for different datasets. We find all read mathces for different datasets with a fixed number of allowed errors. All query datasets of length below 100 bp are allowed to have one error (mismatch or indel) while others are allowed to have two errors. Read matches shown for CloudBurst are both forward

Table 2 StreamAligner performance (times reported in seconds)

Reference genome	Query genome	StreamAligner	SparkBWA	Speedup
S_suis	100 k	18	81	4.5×
chr1	AML	140	54	0.38×
chr21	AML	147	44	0.3×
hg19	AML	87	40	0.46×
chr1	BRL	247	259	1.04×
chr21	BRL	315	322	1.02×
hg19	BRL	121	191	1.57×
chr1	ANL	228	3189	13.98×
chr21	ANL	1222	2403	1.96×
hg19	ANL	337	3369	9.97×
chr1	ERR000589	532	1201	2.25×
chr21	ERR000589	178	407	2.28×
hg19	ERR000589	909	2202	2.42×
chr1	SRR015390	267	1093	4.09×
chr21	SRR015390	191	522	2.73×
hg19	SRR015390	568	2119	3.73×
chr1	SRR062634	779	1819	2.33×
chr21	SRR062634	624	1485	2.37×
hg19	SRR062634	1031	2321	2.25×
chr1	SRR642648	1449	2939	2.02×
chr21	SRR642648	1332	2500	1.87×
hg19	SRR642648	1892	3563	1.88×

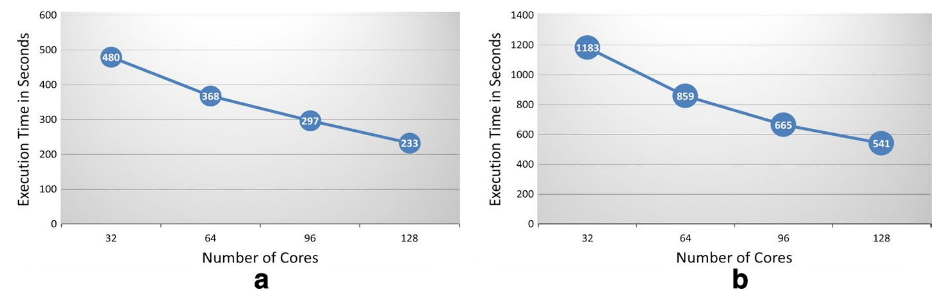

Fig. 10 Execution time for StreamAligner to map (**a**) ANL and (**b**) ERR000589 datasets on Chr1

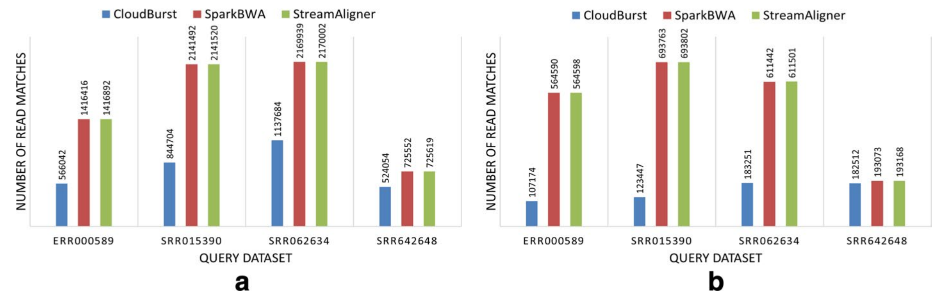

Fig. 11 StreamAligner accuracy evaluation to map different query datasets on (**a**) Chr1 and (**b**) Chr21

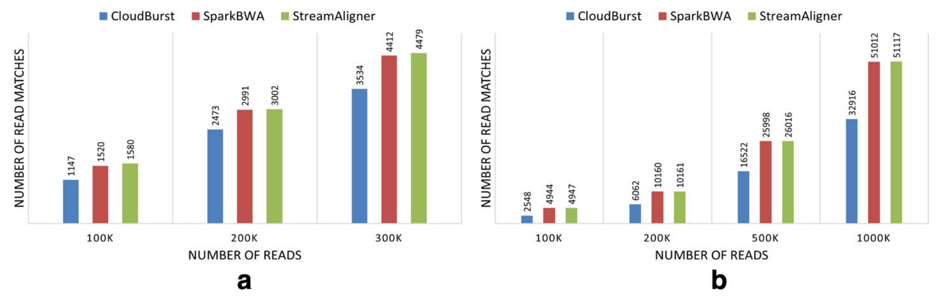

Fig. 12 StreamAligner accuracy evaluation to map (**a**) AML and (**b**) BRL query datasets on Chr1

and backward search matches. SparkBWA is using BWA-SW to find matches with given number of allowed errors.

Figure 11 shows number of read matches found by CloudBurst, SparkBWA and StreamAligner during mapping of different datasets (taken from *1000 genome project*) on Chr1 and Chr21 genome respectively.

Figure 12 shows number of read matches found by CloudBurst, SparkBWA and StreamAligner during mapping of AML and BRL datasets on Chr1.

Conclusions and future work

We have presented a MapReduce based sequence alignment tool called StreamAligner. It has three main features which makes it attractive in comparison to all existing state-of-the-art sequence aligners. Firstly, it aligns stream of reads on reference genome in real-time, hence can be used to automate sequencing and alignment process. The output of sequencing (reads) is given as input to StreamAligner as a stream of reads. StreamAligner aligned stream of reads on reference genome in real-time and store results, therefore avoiding the need to store huge sequencing data. Secondly, it uses suffix array index for read alignment which is generated using distributed index generation algorithm. Due to distributed index generation algorithm, index generation time is very less. Results show that index generation algorithm is highly scalable. Hence, it will generate index in few seconds on large clusters. Third, it needs to upload index only once when StreamAligner is launched. After that index stays in Spark memory and can be used for an infinite time without reloading. Whereas, current state-of-the-art sequence aligner either generate (hash index based) or load (sorted index based) index for every task. Hence, StreamAligner reduces time to generate or load index for every task. A tested version of StreamAligner is available on GitHub with streaming support. StreamAligner is compared to best existing sequence aligners in terms of speed and accuracy and it outperforms all existing sequence aligners. StreamAligner is publicly available to use at GitHub repository: (https://github.com/sanjaysinghrathi/StreamAligner).

In future, the API can be extended to add support for input from different sources like Flume and S3. Further, implementation of StreamAligner on Apache Flink [44], another big data platform will be useful to test the performance comparison.

Authors' contributions
SR performed the literature review, implemented the proposed algorithm and conducted the experiments. AK advised SR all aspects of the paper development. Both authors read and approved the final manuscript.

Authors' information

Sanjay Rathee received the B.Tech degree in computer engineering from Maharshi Dayanand University, Rohtak,Harayana, India, in 2011, and the M.Tech degree in computer engineering from Kurukshetra University, Haryana,India, in 2013. He is currently working toward the Ph.D. degree in computer engineering from Indian Institute ofTechnology, Mandi, India. He has developed several distributed algorithms related to business strategies andbioinformatics sector. His research interests include distributed computing algorithms and platforms, association rulemining and sequence alignment. Arti Kashyap received the B.Sc degree from Himachal Pradesh University, Shimla,H.P., India, in 1989, the M.Sc and Ph.D. degree from Indian Institute of Technology, Roorkee, India, in 1991 and1996 respectively. She is currently working as associate professor at Indian Institute of Technology Mandi, India. Herresearch interests include distributed algorithms, big data analytics, sequence alignment and magnetic materials.

Author details

[1] School of Computing and Electrical Engineering, IIT Mandi, Kamand Campus, Mandi, India. [2] School of Basic Sciences, IIT Mandi, Kamand Campus, Mandi 175005, India.

Acknowledgements
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Availability of data and materials
Not applicable.

Consent for publication
I allow journal to publish it.

Ethics approval and consent to participate
Not applicable.

Funding
Not applicable.

Publisher's Note

Received: 15 November 2017 Accepted: 16 January 2018
Published online: 27 February 2018

References

1. Sabatini R. Ted Talk. (2017). https://www.ted.com/talks/riccardo_sabatini_how_to_read_the_genome_and_build_a_human_being
2. Li H, Ruan J, Durbin R. Mapping short dna sequencing reads and calling variants using mapping quality scores. Genome Res. 2008;18(11):1851–8. https://doi.org/10.1101/gr.078212.108.
3. Cox A. ELAND: Efficient local slignment of nucleotide data (**unpublished**).
4. Jiang H, Wong WH. Seqmap: mapping massive amount of oligonucleotides to the genome. Bioinformatics. 2008;24(20):2395. https://doi.org/10.1093/bioinformatics/btn429.
5. Lin H, Zhang Z, Zhang MQ, Ma B, Li M. ZOOM! Zillions of oligos mapped. Bioinformatics. 2008;24(21):2431–7. https://doi.org/10.1093/bioinformatics/btn416.
6. Smith AD, Chung WY, Hodges E, Kendall J, Hannon G, Hicks J, Xuan Z, Zhang MQ. Updates to the rmap short-read mapping software. Bioinformatics. 2009;25(21):2841. https://doi.org/10.1093/bioinformatics/btp533.
7. Novocraft Technologies Sdn Bhd: NovoAlign. Novocraft Technologies Sdn Bhd. 2008. http://www.novocraft.com.
8. Li R, Li Y, Kristiansen K, Wang J. Soap: short oligonucleotide alignment program. Bioinformatics. 2008;24(5):713–4. https://doi.org/10.1093/bioinformatics/btn025.
9. Campagna D, Albiero A, Bilardi A, Caniato E, Forcato C, Manavski S, Vitulo N, Valle G. PASS: a program to align short sequences. Bioinformatics. 2009;25(7):967–8. https://doi.org/10.1093/bioinformatics/btp087.
10. Eaves HL, Gao Y. Mom: maximum oligonucleotide mapping. Bioinformatics. 2009;25(7):969–70. https://doi.org/10.1093/bioinformatics/btp092.
11. Homer N, Merriman B, Nelson SF. Bfast: an alignment tool for large scale genome resequencing. PLoS ONE. 2009;4(11):1–12. https://doi.org/10.1371/journal.pone.0007767.
12. Kim YJ, Teletia N, Ruotti V, Maher CA, Chinnaiyan AM, Stewart R, Thomson JA, Patel JM. Probematch: rapid alignment of oligonucleotides to genome allowing both gaps and mismatches. Bioinformatics. 2009;25(11):1424–5. https://doi.org/10.1093/bioinformatics/btp178.
13. Malhis N, Butterfield YSN, Ester M, Jones SJM. Slider—maximum use of probability information for alignment of short sequence reads and snp detection. Bioinformatics. 2009;25(1):6–13. https://doi.org/10.1093/bioinformatics/btn565.
14. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.6177. Accessed 15 Mar 2016.
15. Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biol. 2009;10(3):25. https://doi.org/10.1186/gb-2009-10-3-r25.

16. Li H, Durbin R. Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics. 2009;25(14):1754–60. https://doi.org/10.1093/bioinformatics/btp324.
17. Li H, Durbiin R. Fast and accurate long-read alignment with Burrows–Wheeler transform. Bioinformatics. 2010;26(5):589. https://doi.org/10.1093/bioinformatics/btp698.
18. Ferragina P, Manzini G. Opportunistic data structures with applications. In: Proceedings of the 41st annual symposium on foundations of computer science. FOCS '00, IEEE computer society, Washington, DC; 2000. p. 390. http://dl.acm.org/citation.cfm?id=795666.796543
19. Apache Hadoop. http://hadoop.apache.org/.
20. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on hot topics in cloud computing. HotCloud'10. USENIX Association, Berkeley; 2010. p. 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113.
21. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on operating systems design & implementation, Vol 6. OSDI'04. USENIX Association, Berkeley, CA, USA; 2004. p. 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264.
22. Schatz MC. Cloudburst: highly sensitive read mapping with mapreduce. Bioinformatics. 2009;25(11):1363–9. https://doi.org/10.1093/bioinformatics/btp236.
23. Nguyen T, Shi W, Ruden D. Cloudaligner: a fast and full-featured mapreduce based tool for sequence mapping. BMC Res Notes. 2011;4:171. https://doi.org/10.1186/1756-0500-4-171.
24. Schatz MC. BlastReduce: high performance short read mapping with MapReduce.
25. Abuin JM, Pichel JC, Pena TF, Amigo J. BigBWA: approaching the Burrows-Wheeler aligner to Big Data technologies. Bioinformatics. 2015;31(24):4003–5. https://doi.org/10.1093/bioinformatics/btv506.
26. Abuin JM, Pichel JC, Pena T, Amigo J. Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. PLoS ONE. 2016;11(5):1–21. https://doi.org/10.1371/journal.pone.0155461.
27. Decap D, Reumers J, Herzeel C, Costanza P, Fostier J. Halvade: scalable sequence analysis with mapreduce. Bioinformatics. 2015;31(15):2482–8. https://doi.org/10.1093/bioinformatics/btv179.
28. Congosto M, Basanta-Val P, Fernández LS. T-hoarder: a framework to process twitter data streams. J Netw Comput Appl. 2017;83:28–39.
29. Lv Z, Song H, Basanta-Val P, Steed A, Jo M. Next-generation big data analytics: state of the art, challenges, and future research topics. IEEE Trans Ind Inf. 2017;13(4):1891–9. https://doi.org/10.1109/TII.2017.2650204.
30. Rathee S, Kaul M, Kashyap A. R-apriori: an efficient apriori based algorithm on spark. In: Proceedings of the 8th workshop on Ph.D. workshop in information and knowledge management. PIKM. ACM, Melbourne, Australia. 2015;15: 27–34. https://doi.org/10.1145/2809890.2809893
31. Basanta-Val P, Fernández-García N, Basanta-Val P, Fernández-García N, Sánchez-Ferná ndez L, Arias-Fisteus J. Patterns for distributed real-time stream processing. IEEE Trans Parallel Distrib Syst. 2017;28(11):3243–57. https://doi.org/10.1109/TPDS.2017.2716929.
32. Basanta-Val P, Fernández-García N, Wellings AJ, Audsley NC. Improving the predictability of distributed stream processors. Future Gener Comput Syst. 2015;52(C):22–36. https://doi.org/10.1016/j.future.2015.03.023.
33. Pireddu L, Leo S, Zanetti G. Seal: a distributed short read mapping and duplicate removal tool. Bioinformatics. 2011;27(15):2159–60. https://doi.org/10.1093/bioinformatics/btr325.
34. Li Y, Zhong S. Seqmapreduce: software and web service for accelerating sequence mapping. Critical assessment of massive data analysis (CAMDA). 2009.
35. Langmead B, Schatz MC, Lin J, Pop M, Salzberg SL. Searching for snps with cloud computing. Genome Biol. 2009;10(11):134. https://doi.org/10.1186/gb-2009-10-11-r134.
36. Amazon EC2 Cloud: Amazon Web Services Genomics. Amazon EC2 Cloud. 2017. https://aws.amazon.com/health/genomics/.
37. Versaci F, Pireddu L, Zanetti G. Scalable genomics: from raw data to aligned reads on apache yarn. EEE Int Conf Big Data. 2016;12:1232–41. https://doi.org/10.1109/BigData.2016.7840727.
38. Peters D, Qiu K, Liang P. Faster short dna sequence alignment with parallel bwa. AIP Conf Proc. 2011;1368(1):131–4. https://doi.org/10.1063/1.3663477.
39. Klus P, Lam S, Lyberg D, Cheung MS, Pullan G, McFarlane I, Yeo GS, Lam BY. Barracuda—a fast short read sequence aligner using graphics processing units. BMC Res Notes. 2012;5(1):27. https://doi.org/10.1186/1756-0500-5-27.
40. Zhou W, Li R, Yuan S, Liu C, Yao S, Luo J, Niu B. Metaspark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes. Bioinformatics. 2017;33(7):1090. https://doi.org/10.1093/bioinformatics/btw750.
41. National Center for Biotechnology Information. Reference Genomes. National Center for Biotechnology Information. 2017. http://www.ncbi.nlm.nih.gov.
42. Agilent Inc. USA. Query Datasets. Agilent Inc. USA. 2017. https://cloud.iitmandi.ac.in/d/2ba909564a/.
43. 1000genomes Project: Genome Dataset Project. 1000genomes Project. 2017. ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/.
44. Apache: Flink. Apache. 2017. https://flink.apache.org/.