# Next Generation Indexing for Genomic Intervals

Vahid Jalili, Matteo Matteucci, Jeremy Goecks, Yashar Deldjoo, and Stefano Ceri

**Abstract**—Di4 (1D intervals incremental inverted index) is a multi-resolution, single-dimension indexing framework for efficient, scalable, and extensible computation of genomic interval expressions. The framework has a tri-layer architecture: the *semantic layer* provides orthogonal and generic means (including the support of user-defined function) of *sense-making* and higher-lever reasoning from region-based datasets; the *logical layer* provides building blocks for region calculus and topological relations between intervals; the *physical layer* abstracts from persistence technology and makes the model adaptable to variety of persistence technologies, spanning from small-scale (e.g., B+tree) to large-scale (e.g., LevelDB). The extensibility of Di4 to application scenarios is shown with an example of comparative evaluation of ChIP-seq and DNase-Seq replicates. Performance of Di4 is benchmarked for small and large scale scenarios under common bioinformatics application scenarios. Di4 is freely available from https://genometric.github.io/Di4.

**Index Terms**—Index structures; efficient query processing; genomic data management

## 1 Introduction

The third paradigm shift in genome sequencing technologies—real-time, single-molecule—is emerging, upending the field after electrophoretic (first) and massively parallel or next generation sequencing (NGS, second) paradigm shifts. These technological improvements diminish genome sequencing cost (from \$100M per genome in 2001 to \$1K in 2015, with expected further drop to \$100 soon) and expedite sequencing time (e.g., Oxford Nanopore yield data within 30min of sample application), thereby enabling "universal monitoring" of nucleic acids [1]. Due to these technological advances, we are approaching the milestone where genome of 0.1% of living humans are sequenced to some extent [1]. This emphasizes the explosive growth in genomic data production and application [2], which may soon become the biggest and most important *big data* problem of humanity [3]. In this paper we discuss a holistic information retrieval framework which provides building blocks for a scalable and transparent *sense-making* from genomic datasets.

### 1.1 A genomics primer

The procedure of a genome sequence analysis can be defined in three steps; primary, secondary, and tertiary analysis [4]. Primary analysis is concerned with genome sequencing, producing short *reads* of four nucleotides (i.e., Adenine (A), Guanine (G), Cytosine (C) and Thymine (T) in DNA, or Uracil (U) in RNA). Secondary analysis is concerned with assembling or aligning the sequenced DNA/RNA fragments and building a whole genome representation, which is then analyzed for feature extraction (e.g., determination of variations). Tertiary analysis is concerned with *making sense* from the extracted features, e.g., discovering how heterogeneous regions (i.e., regions of independent experiments identifying genomic characteristics with different markers) interact with each other; it is attracting increasing interest, as huge datasets produced by secondary analysis are made available by large international consortia (such as ENCODE (encodeproject.org), TCGA (cancergenome.nih.gov), and 1000 Genomes Project (internationalgenome.org)).

### 1.2 The challenge

While genomic data is generally abstracted as sequences of nucleotides at primary and secondary analysis, tertiary analysis commonly describes genomic data in the form of regions of DNA, because these contiguous stretches of nucleotides have known biological functions, such as coding for proteins or serving as binding sites for proteins. Region-based, genome-wide datasets include variations (e.g., modifications, insertions, or deletions at given DNA positions), signals (e.g., measures of transcriptional activity), peaks (e.g., regions with higher DNA read density with respect to the background signal), or structural properties of the DNA (e.g., break points where the DNA is damaged, or junctions where DNA creates loops).

The challenges of information retrieval from genomics interval-based data can be studied from three facets; first, while each dataset describes a single biological experiment, it is the comparative assessment of datasets that enable studies such as precision medicine, or drug response prediction. However, comparative assessment of large datasets (e.g., UK Biobank with 500,000 participants, the largest human genetic dataset) is a massive operation that requires novel approaches for genomic interval operations. Second, a problem-driven explorative approach for *making sense* of data during tertiary analysis commonly leads dry-lab scientists to roll proprietary and ad-hoc solutions, typically by integrating existing "building blocks". This highlights the need for a generic, comprehensive, extensible, and orthogonal region calculus for genomic intervals. Third, the runtime of tertiary analysis is marginal to that of primary and secondary analysis, however, the primary and secondary analysis operations are commonly run only once on a given data, while tertiary analysis operations are executed frequently

- *Dip. di Elettronica, Informazione e Bioingegneria (DEIB) – Politecnico di Milano. 20133 Milano, Italy*
*E-mail: vahid.jalili@polimi.it*

*Manuscript received XXXX; revised XXXX*

for exploration and *sense-making*, which highlights the need for an agile query execution framework.

While many solutions for efficient data management of "sequence reads" have been developed, this manuscript concentrates on efficient data management for genomic intervals. We present **Di4** (1D intervals incremental inverted index), a multi-resolution single-dimension indexing framework over interval-based NGS data. Di4 aggregates concepts from spatio-temporal databases, H264 video encoding, and signal processing to deliver a high-end indexing framework for genomics, with the objective of facilitating efficient *sense-making*.

### 1.3 State of Art

We organize the state of the art by first illustrating the foundations and limitations of methods for region-based computation, then common practices in bioinformatics and related studies in temporal databases.

Classical search trees such as interval trees [5], segment trees [6], range trees [7], or Fenwick trees [8] are optimal solutions each for particular interval-based retrieval, and some are used in common bioinformatics tools as underlying data structure (e.g., UCSC Genome Browser uses R-Trees [9]). However, such data structures do not natively provide a comprehensive solution for tertiary analysis challenges. For instance, the query "find all the intervals intersecting a given interval" can be solved in $O(\log n + m)$ (where $n$ is the number of intervals in the tree, and $m$ is the number of intervals returned at a query execution) using an interval tree, but the query "find $n$-th closest intervals" requires defining a new method for traversing the interval-tree. As another example, R-tree partitions intervals into hierarchical *bins*, hence non-uniformly distributed intervals (which are common in genomic datasets such as ChIP-seq, RNA-seq, and exome sequencing) unbalance bin loads; consequently, some bins take considerably longer time to be processed than others.

Some array storage technologies are adapted to store/query genomic data. Among them, the tool TileDB [10] persists NGS data, and queries them through its wrapper called GenomicsDB. Despite of promising performance in persisting and querying data, array-based approaches fail to support general purpose NGS data querying needs due to the choice of specific array dimensions. For instance, TileDB stores single-nucleotide polymorphisms (SNPs) in either column or row storage formats, chosen at initialization time. If the former is chosen, queries can efficiently compute the intersection of SNPs, but queries for regions with a particular number of SNPs require a linear scan. If latter is chosen, queries for SNPs belonging to a sample are efficiently supported, but querying for the intersection of SNPs requires a linear scan on the whole array. However, random access to array columns/rows is an incomplete region calculus, which is limited in computing other typical bioinformatics functions, e.g., calculating the Jaccard index of datasets.

NGS machines produce files each referring to a biological experiment, and bioinformatic pipelines apply to input files yielding output files; therefore many bioinformatics tools and environments operate on data stored in plain text format on file systems. A drawback of leveraging on such file systems is the penalty of sequential accesses. For instance, the query "given a region, find overlapping regions across all the files" can be answered by linearly scanning all the files independently and in parallel. However, queries such as "find regions where at least $80\%$ of the files have an overlap" (similar to querying Fenwick trees [8]) requires scanning all the files together. There has been efforts to provide random access to NGS interval-based files, e.g., BITS [11] returns a "count" of intersection, similar to the cardinality of the output of querying an interval tree [5], however, returning only the count of intersection is insufficient to execute typical bioinformatics functions on interval-based genomic data.

The bioinformatics community offers some tools for particular querying purposes, such as GEMINI [12] which leverages SQLite to provide retrievals on genetic variations. It is mainly designed to explore mutational burden in pathways and interacting proteins. Additionally, GEMINI leverages basic SQLite functions to execute queries such as "how many heterozygote are observed for a given variation" (similar to TileDB and BITS queries). However, such systems are tailored for particular querying purpose, and cannot be extended to support the wider querying demands of NGS data *sense-making* challenges.

The bioinformatics community attempted to define region calculus building blocks, and offers tools such as BEDTools [13] and BEDOPS [14]. These tools are widely accepted by the community and are used in both systemic solutions (e.g., Galaxy [15]) and ad-hoc pipelines. However, the functions implemented in such tools are mainly designed for ad-hoc solutions, and do not scale efficiently *w.r.t.* the large-scale and growing genomic datasets. Accordingly, Di3 [4], an indexing framework with building blocks for querying big genomics data, and Giggle [16], a large scale similarity search tool, are developed. A detailed discussion is presented in Section 3.

### 1.4 Our contribution

The most important contribution of Di4 is its extendable, orthogonal, and comprehensive region calculus. In spite of focusing on the genomic domain, Di4 is designed for any domain that provides a comparer for the chronological order of its elements and an operator for absolute distance between any two elements.

Di4 design is coherent with three major design decisions. First, the framework is defined at data access layer, independently from business logic and data layer, and adaptable to any underlying key-value pair persistence technology (spanning classical data structures such as B+ tree, or a cloud-based B+ tree [17], to LevelDB (github.com/google/leveldb) and Monkey [18], according to the architecture in Figure 1). Such separation makes Di4 adaptable to a variety of application scenarios from small scale ad-hoc solutions (using B+ tree), to large scale systemic solutions (using cloud-based key-value pair persistence technologies, e.g., LevelDB or Monkey [18]). Second, the framework is extensible, as it has a modular definition of functions, where each of them accepts user-defined functions (UDF) through behavioral design patterns, such as the *strategy pattern* (see Figure 1, and Section 2.2 for deeper discussion). Third,

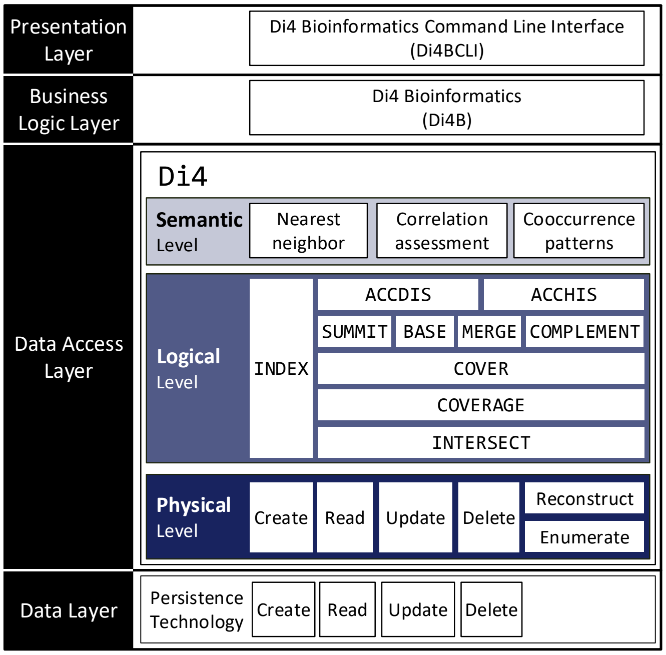


Fig. 1: Di4 architecture and functional components.

it adopts a multi-resolution design to optimize querying data with sargable and non-sargable criteria. Its primary resolution indexes NGS intervals by coordinate attributes, and its secondary resolutions use PDF-optimized scalar quantization to heuristically optimize non-sargable queries (deeper discussion postponed to Section 2.3.4).

Di4 improves over Di3 [4], which stores all the intervals overlapping the left or right-end of an interval on the genome; conversely, Di4 recursively infers this information from neighbor regions. Additionally, Di4 benefits from signal scalar quantization methods to efficiently load-balanced parallelization and implements an effective heuristic for decreasing the number of elements to be processed when executing a query. As a consequence, Di4 is faster than Di3 in retrieving from the index (deeper discussion is postponed to Section 3.3), and also faster than BEDTools, BEDOPS, and Giggle (benchmarked in Section 3).

In the following, first the interval-based abstraction of genomic regions is explained, then the Di4's approach for modeling these intervals is discussed. Querying genomic intervals leveraging Di4's model is subsequently explained in three layers of abstractions. The method of organizing intervals in Di4's model is discussed in sections 2.3 and 2.4. Finally, Di4 is benchmarked against state-of-the-art in Section 3.

## 2 Method

The genome consists of nucleic acid sequence—a succession of four nucleotides: A, C, G, and T/U—and it is commonly modeled as a linear (unbranched) and one-dimensional succession of A, C, G, and T/U letters.

A position on genome is commonly referenced with three methods; first, the nucleotide sequence of the position (see panel C on Figure 2). Second, represent three consecutive nucleotides (codon) by a letter (i.e., amino acid code) associated with the corresponding proteinogenic amino acid (see panel D on Figure 2). A third approach is to use the coordinates of a position on a genome; commonly referenced as *chromosome*, *start* and *stop* positions (see panel E on Figure 2), which is commonly associated with a set of metadata for the position (e.g., a p-value of a DNA-protein binding significance).

Each of these reference types is used in various genome data analysis [4]. The data indexing framework discussed in this manuscript (Di4) is defined over the interval representation. Accordingly, in this section we provide a conceptual description of the Di4 data model, including its data structures and operations.

### 2.1 Di4 Data Model

Consider a continuous domain with an order relation, i.e., an arbitrary element $e_A$ proceeds/succeeds element $e_B$. Let us represent a *durative* event on the domain with three attributes: (i) *start*, a single point-in-domain where the action begins, (ii) *stop*, a single point-in-domain where the action is accomplished, and (iii) *middle*, an infinite sequence of points-in-domain where the action is being executed; such that, a durative event is happening between an *inclusive* start and *exclusive* stop. Events are commonly modeled as intervals on a domain, with start and stop of the event being respectively the left and right-end of the interval. Additionally, an interval describes an events using its metadata (e.g., the p-value of a ChIP-seq peak, or reference and alternative alleles of a variation).

Di4 leverages the research in the field of temporal databases and multi-dimensional data structures (surveyed in [19], [20]), and augments the snapshot index [21], [22] and the organization of time index on a tree data structure [23], [24], to model genomic intervals using snapshots. A *snapshot* is a key-value pair object $B_b$ which *bookmarks* a position on a domain by capturing coordinate characteristics, overlapping intervals, and their relative behavior (see Section 2.3). Snapshots bookmark intervals leveraging the *instantaneous model* assumption according to which any intervals on a continuous domain can be explicitly represented using just its *start* and *stop* attributes, and the *middle* attribute of durative events is represented implicitly. Let us consider,

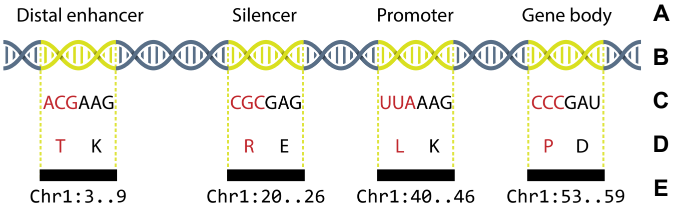


Fig. 2: A synthetic example of various genomic data representation methods. Genome is represented linearly by chromosomes; a chromosome is a DNA molecule (B) that has functional units (A), which are commonly referenced using nucleic acid sequence (C), protein sequence (D), or intervals referring to the first and last base-pair of a regions-of-interest (E). These methods (C, D, and E) are commonly used to represent various genomic activities such as DNA-protein interaction or variations.

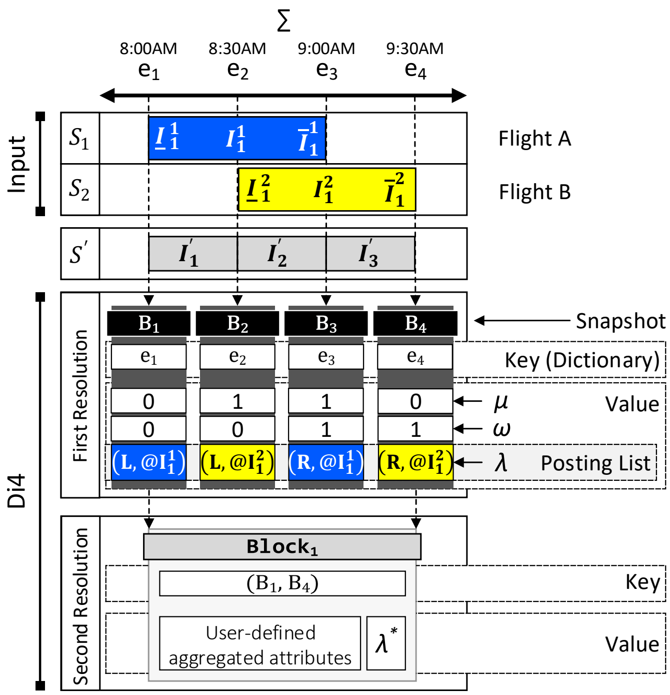


Fig. 3: Di4 notation and data structure. Posting lists denote *causal* intervals, for instance, $I_1^1$ at $\lambda_1$. $\mu$ is the number of overlapping *non-causal* events, for instance $\mu_2 = 1$ because $I_1^1$ is a non-causal event for $B_2$. $\omega$ is the number of *causal* and *stopping* events, for instance, $\omega_4 = 1$ because $I_1^2$ stops at $e_4$.

for instance, the two “Flight” events of Figure 3 modeled using four snapshots as follows:

- $B_1$ *8:00AM*. **Explicit**: Flight-A departs (causal event).
- $B_2$ *8:30AM*: **Explicit**; Flight-B departs (causal event). **Implicit**: Flight-A is flying, and was flying between *8:00AM* and *8:30AM*.
- $B_3$ *9:00AM*: **Explicit**; Flight-A lands (causal event). **Implicit**: both flights were flying between *8:30AM* and *9:00AM*.
- $B_4$ *9:30AM*: **Explicit**; Flight-B lands (causal event). **Implicit**: Flight-B was flying between *9:00AM* and *9:30AM*.

In general, a snapshot on the domain has at least one *causal event*, and any number of *non-causal events*. The *causal-event* and *non-causal-events* of a snapshot, are respectively the events whose start/stop, or middle attribute is bookmarked by the snapshot. For instance, the causal even of snapshot $B_2$ is the “depart of Flight-B”, and its non-causal event is “Flight-A is flying“. Snapshots bookmark causal events *explicitly* by pointers to the events. For instance, the snapshot $B_2$ has a pointer to the “Flight-B” in its *posting list* (see Figure 3; discussed in details in Section 2.3); a pointer could be, for instance, the ID of a flight in a database containing all the related information such as the passenger list.

To bookmark events with a snapshot, a pointer to a causal event is required, while pointers to non-causal events can be inferred from neighbor snapshots; hence storing those is normally suboptimal and redundant. Di4 adopts an *incremental inverted index* paradigm where the pointers to non-causal events are not stored. Snapshots represent non-causal events *implicitly* by keeping track of their count only (using the $\mu$ component of a snapshot; discussed in details in Section 2.3); for instance, in the example of Figure 3, the snapshot at *8:30AM* reports “Flight-B departs and one other flight is flying” ($\mu_2 = 1$), without knowing that the other flight is “Flight A” (i.e., no pointer to the “Flight A” is present in the posting list of the snapshot).

## 2.2 Di4 Information Retrieval and Inference

Di4 retrieval functions are defined at three levels, *physical*, *logical*, and *semantic*, as described in Figure 1. The functions of each layer are defined leveraging the functions of the layers beneath it (and physical layer leverages data layer application programing interface (API)).

The semantics of the Di4 retrieval functions has been divided between internal and external semantics. The internal semantic is a function-specific logic, and the external semantic is an application-specific logic provided to the function as a procedural parameter (aka user-defined function (UDF)). The internal and external semantics are integrated, which allows manipulation of intermediate steps of the function by the external semantics. Such design keeps Di4 retrieval functions at an abstract level, while still applicable to any application specific scenarios. MuSERA, a tool for reproducibility assessment across ChIP-seq replicates which is based on the Di3 index [25], uses these retrieval functions as building blocks to identify consensus peaks across ChIP-seq replicates [26], to assess the correlation of replicates, to find the distance distribution of nearest neighbors on functional genome positions, and to implement a genome browser.

Table 1 presents sample application scenarios which can be implemented by augmenting Di4 retrieval functions with UDFs, which the present section explains Di4 retrieval functions (building blocks) and their integration with UDFs.

### 2.2.1 *Low-level (physical level)*

Physical level functions bridge the Di4 data model to the data layer. The operations provided by the physical level, are low-level operations spanning *Create*, *Read*, *Update*, *Delete* (CRUD), *Enumerate*, and *Reconstruct* (see Figure 1) leveraging API of the actual data layer technology. These operations create and manipulate the snapshots and organize them in a key-value pair storage, by translating input intervals into snapshots, and retrieving and *reconstructing* intervals from snapshots (see Section 2.3.3). They are internal to Di4 and, accordingly, do not incorporate UDFs.

### 2.2.2 *Mid-level (logical level)*

Logical level functions leverage physical level operations, and they yield the essential elements for region calculus using snapshots.

These functions stem from the co-occurrence of intervals, such that they either (a) find indexed intervals co-occurring a given interval, (b) find co-occurring indexed intervals which satisfy a criterion, or (c) retrieve an (aggregated) attribute of co-occurring indexed intervals.

Two events are called *co-occurring* if they are co-localized on the domain (i.e., the distance between them is constrained, but not necessarily set to zero). We adopt a definition that keeps into account the following aspects. First,

| Function | Description | Application Example |
|---|---|---|
| INTERSECT | Find index intervals co-occurring with a given interval. | How many COSMIC variants appear in c-Myc transcription factor binding regions? |
| COVER | Find regions on domain where a particular number of intervals are co-occurring. | Find those sites related to H3k4me3 modification where a significant (p-value < 1E-8) DNA-protein binding is observed in at least 10 samples, and their combined significance at each site, using Fisher's method, is more stringent than 1E-10. |
| ACCHIS | Computer histogram of accumulation on entire domain or selected areas. | Intersect and create histogram of all cancer variants in COSMIC vs. ENCODE annotations. The goal would be to understand if some transcription factor binding sites are subject to mutation more often than others are. |
| NEAREST NEIGHBOR | Find indexed intervals at a given proximity to a given reference interval. | Determine a distance distribution between indexed enriched regions and a given set of peaks, which could indicate how close the determined binding sites are to known genomic features. |
| CORRELATION ASSESSMENT | Find Jaccard index between reference and indexed intervals, computed as a ratio between the number of overlapping genomic bases and the total number of bases. | Find how similar (in terms of Jaccard index) the determined enriched regions are to c-Myc transcription factor binding sites. |

TABLE 1: Sample application scenarios for a subset of Di4 retrieval functions.

**Algorithm 1** INTERSECT function; it finds intervals overlapping or at $d$ distance of reference intervals $\{I_r\}$, and passes them to a UDF ($U$).

```
procedure INTERSECT($\{I_r\}, d, U$)
  for each $I_r$ do
    block ← find a block whose left-end is closest on the right of $\underline{I}_r - d$
    $B_b$ ← find a snapshot whose coordinate is closest on the right of $\underline{I}_r - d$
    $i \leftarrow 0$
    do
      OPEN($B_{b+i}$, block)                                    ▷ see Algorithm 4
      $i \leftarrow i + 1$
      while $e_{b+i} < \bar{I}_r + d$ do
        INRECONSTRUCT($B_{b+i}$)                                ▷ see Algorithm 4
        $i \leftarrow i + 1$
      while false = canClose ← EXRECONSTRUCT($B_{b+i}, U, \langle I_r, d \rangle$) do
        $i \leftarrow i + 1$
        if $B_{b+i}$ overlaps $I_{r+1}$ with $d$ distance proximity then
          $r \leftarrow r + 1$
          break
    while canClose = false
```

the location of events in some applications could be approximated; for instance, in genomics, the location of a *peak* on ChIP-seq data could be considered with ±10base-pair approximation. Second, co-occurrence could be studied on coarse granularity; for instance, in genomics, two intervals, one on an enhancer, and another on a related gene transcription start site, might be considered co-occurring when they are at a given distance from each other (e.g., at $340k$base-pair [27]).

The first function is INTERSECT, which is based on the co-occurrence of intervals and covers classical region calculus (see Algorithm 1). For instance, "given a point/interval on the domain, find all intervals overlapping with it", similar to the queries on interval trees [5] and segment trees [6]. Note that the INTERSECT function considers two intervals overlapping if they are $d$ distance apart.

A common inference on spatial, temporal, and spatiotemporal data, is the check for events compliance with a particular property or function $f$; this is commonly known as *coverage* on $f$. This analysis has application-specific definition and criteria. For instance, "find positions on genome where a test statistic calculated by combining p-values of co-occurring intervals using Fisher's method, is more stringent than $1\text{e}{-8}$". Accordingly, Di4 defines a COVERAGE function which finds a proportion of the domain which contains snapshots all evaluated as true value of interest defined by function $f$. The semantic of COVERAGE is partially determined by function $f$, which is a UDF. The Di4 can analyze for *coverage* on $f$ both on the entire domain and specific positions with $d$ distance proximity. Note that, the criteria of function $f$ could be defined on indexed or non-indexed attribute of intervals (sargable or non-sargable). Accordingly, Di4 leverages its *secondary resolutions* to heuristically improve executing *coverage on* $f$ for non-sargable attributes (e.g., see Algorithm 2 and Section 2.3.4).

Genomics is commonly interested in "coverage of accumulation" (i.e., $f$ := *accumulation*). *Accumulation* is the number of intervals overlapping a certain point on the domain. For example, as more intervals are found to include a particular variant, the higher the confidence that the variant is real and not a sequencing artifact. Accordingly, the *coverage of accumulation* function, COVER, yields a set of consecutive snapshots whose referenced intervals are of a specific accumulation (see Algorithm 2). The function leverages two aggregated attributes of snapshots encapsulated by secondary resolution blocks (see Section 2.3.4) to minimize the number of snapshots to be traversed; the attributes are $g_{\text{min}}$ and $g_{\text{max}}$, which are the minimum and maximum accumulation at snapshots encapsulated by secondary resolution blocks. Note that, blocks can store any user-defined aggregated attribute(s) of intervals/snapshots; accordingly, $g_{\text{min}}$ and $g_{\text{max}}$ are an example of aggregated attributes used for *coverage on accumulation* (i.e., COVER) function.

In the simplest setup of the COVER function, Di4 implements MERGE and COMPLEMENT functions which are respectively the coverage of at "least one" and "zero" accumulation. Additionally, Di4 defines SUMMIT and BASE functions, which respectively maximize and minimize the COVER function. In other words, they find regions on the domain with local maximum or minimum accumulation within a given range.

Note that the logical level functions incorporate a UDF and can be applied on the entire domain or the specific positions with $d$ distance proximity. This design makes the functions extremely extensible. For instance, consider the following query:

> *(a) find promoter regions which are covered by at least 3 overlapping intervals within a* $1kbp$ *proximity, (b) where the p-value of each interval is more stringent than* $1\text{e}{-4}$ *, (c) and their combined p-value using Fisher's method is more stringent than* $1\text{e}{-8}$.

The section (a) of this query defines portions on the genome where a COVER function should search for the accumula-

**Algorithm 2** COVER function; it finds regions on domain where at least $a_{\text{min}}$ and at most $a_{\text{max}}$ intervals overlap, and passes them to a UDF ($U$).

```
procedure COVER($a_{\text{min}}, a_{\text{max}}, U$)
  for each block in secondary resolution do
    if $a_{\text{max}} < g_{\text{min}}$ or $g_{\text{max}} < a_{\text{min}}$ then
      continue
    $a_{\text{tag}} \leftarrow -1$; $b_{\text{tag}} \leftarrow -1$; $i \leftarrow 1$
    $B_b \leftarrow$ first snapshot encapsulated by the block
    if $a_{\text{min}} \leq$ accumulation at $e_{b+i} \leq a_{\text{max}}$ then
      $a_{\text{tag}} \leftarrow$ accumulation at $e_b$; $b_{\text{tag}} \leftarrow b$
      OPEN($B_b$, block)                                   ▷ see Algorithm 4
    do
      $i \leftarrow i + 1$
      if $a_{\text{tag}} = -1$ & $a_{\text{min}} \leq$ accumulation at $e_{b+i} \leq a_{\text{max}}$ then
        $a_{\text{tag}} \leftarrow$ accumulation at $e_{b+i}$; $b_{\text{tag}} \leftarrow b$
        OPEN($B_b$, next block)                            ▷ see Algorithm 4
      else if $a_{\text{tag}} \neq -1$ then
        if accumulation at $e_{b+i}$ not in range $[a_{\text{min}}, a_{\text{max}}]$ then
          $a_{\text{tag}} \leftarrow -1$
          while false = canClose ←EXRECONSTRUCT($B_{\text{tag}}, U, \langle b_{\text{tag}}, b+i \rangle$) do
            $i \leftarrow i + 1$
            if $a_{\text{min}} \leq$ accumulation at $e_{b+i} \leq a_{\text{max}}$ then
              break
        else
          INRECONSTRUCT($B_{b+i}$)                          ▷ see Algorithm 4
    while canClose = false
```

tion of at least 3 intervals (internal logic). The section (b) defines a criteria for counting the accumulation (internal logic and UDF). The section (c) manipulates the output of the `COVER` function and returns the promoter region if the combined p-value of the overlapping intervals is more stringent than $1\text{e}{-8}$, which is a logic defined by a UDF. This query defines a comparative enrichment assessment of genomic intervals—a daily-based analysis in genomics pipelines, and it is partially an application-specific query; however, still it can be implemented using Di4 without altering the functions due to the internal and external logic (UDF) integration of the functions.

Di4 also defines functions for statistical summary of data; the functions are based on the `COVERAGE` function, and summarize accumulation as histogram (`ACCHIS`) and frequency (`ACCDIS`) distribution.

### 2.2.3 High-level (semantic level)

Upon physical level operations and logical level functions, Di4 builds semantic level functions. The goal of these functions is to facilitate high-level reasoning on data. These functions are based on coordinate attributes, provide first subjective impression on the data, and, through UDF, allow further application-specific processing. In the following we briefly discuss some of these functions.

Co-occurrence patterns: A co-occurrence pattern represents a subset of samples whose intervals are frequently co-localized on the domain. Genomics is interested in both co-occurrence (only coordinate attribute) and mixed-feature (coordinate and additional attributes) patterns. The later is well-studied as *mixed-drove* co-occurrence pattern mining, where patterns are commonly identified in multiple steps, that is by identifying mixed-drove candidate patterns on one attribute based on contributing or non-contributing (false-candidates) intervals, and pruning-out the candidates by patters of other attributes. Di4 adapts to mixed-drove co-occurrence pattern mining by identifying quantity-based co-occurrence patterns on coordinate attributes, and incorporating user-defined application-specific pattern finding method on additional attributes via UDF.

**Algorithm 3** Nearest neighbor function; it finds nearest neighbors to the given point $e$ on domain which satisfies a user-defined criteria, $U$, and returns the nearest neighbors or their distance depending on the $d$ argument.

```
procedure NEAREST_NEIGHBOR($e, d, U$)
  find $B_b$ where $e_{b-1} < e \leq e_b$
  $i \leftarrow 0$
  do
    $i \leftarrow i + 1$
    $d \leftarrow -1$
    if $U$ ({intervals bookmarked by $B_{b-i}$}) = true then
      $d \leftarrow e_b - e_{b-i}$
    if $U$ ({intervals bookmarked by $B_{b+i}$}) = true then
      $d \leftarrow \min(d, e_{b+i} - e_b)$
  while $d \neq -1$
  if $t$ = true then
    return $d$
  else
    return intervals bookmarked by $B_d$
```

Nearest neighbor: Genome is commonly modeled as a single dimension domain (*chromosome, start, stop*), and it differentiates between *up-stream* (preceding) and *down-stream* (succeeding) neighbors of a given reference interval. Di4 determines nearest neighbors based on two distance metrics: chronological order ($n$-th closest neighbor), and absolute distance (neighbor at maximum $d$ distance). To find neighbors, Di4 first finds the pivot snapshot (reference point), and processes its up- and down-stream neighbor snapshots based on the distance metric, to return the intervals bookmarked by the determined snapshots (see Algorithm 3). For instance, it can execute queries such as "find nearest position on domain to a given $e$ point, where the position is a *promoter* region with at least 3 overlapping intervals each with *p-value* $< 1\text{e}{-8}$". It requires $O(\log_b n)$ (when a B+ tree is used as persistence technology, for a blocking factor $b$ and $n$ number of snapshots) to find a pivot snapshot, and $O(1)$ to access each of its neighbors (regardless of the distance metric); therefore, the asymptotic performance of Di4 for this operation is $O(\log_b n)$. The pseudocode of nearest neighbor function is given in Algorithm 3.

Correlation assessment: Similar to co-occurrence patterns, correlation is also an attribute-dependent function. Therefore, Di4 takes a similar approach to co-occurrence patterns by defining correlation based on coordinate attribute, and enabling a UDF to process additional attributes. Di4 uses Jaccard index to determine a coordinate-based correlation coefficient; it finds the regions of intersection and union using the functions `SUMMIT` and `MERGE` respectively.

## 2.3 Di4 Data Indexing

Di4 adopts a multi-resolution approach for interval indexing. At the *first resolution* Di4 takes snapshots of events and stores minimal essential information for data integrity, accuracy, and consistency. Data in the first resolution are then aggregated into the *second resolution* layer which aggregates the information of first resolution to heuristically prune the number of snapshots to be scanned for specific queries and speed up search and retrieval. In the following we describe the details of first and second resolution indexing.

#### 2.3.1 First Resolution Data Structure

Let $\Sigma$ denote the domain (i.e., the universe of all elements constituting intervals) and $e \in \Sigma$ any element of such domain. $I = [\underline{I}, \bar{I})$, $\underline{I} < \bar{I}$ denotes an interval with $\underline{I} \in \Sigma$ and $\bar{I} \in \Sigma$ stating respectively the start (left-end) and stop (right-end) of interval $I$. An interval $I$ is then a left-closed and right-open interval of ascending ordered pair of $e$ elements.

Intervals referring to a common phenomenon are organized in sets, or *samples* (e.g., all regions produced by a given experimental condition), denoted as $\mathbb{S} := \{S_1, \dots S_j, \dots S_J\}$ where $S_j := \{I_1^j, \dots I_i^j, \dots I_{|S_j|}^j\}$.

The superimposition of intervals given by input samples ($\mathbb{S}$) induces a new set of non-overlapping intervals on the domain, denoted by $S'$ (see Figure 3). The intervals of $S'$ dichotomize the domain, and form the basis of the first resolution index. Let $I'$ denote an interval of a new set $S' := \{I'_1 \dots I'_i \dots I'_{|S'|}\}$, where the coordinates of $I'_i$ are defined by the input intervals. For instance, referring to Figure 3, the left and right ends of $I'_1$ is defined respectively by the left ends of intervals $I_1^1$ and $I_1^2$.

The first resolution of Di4 implements the essential aspects of the model through an incremental inverted index where each unique point on the domain ($e_i$) defined by the left or right ends of $I'_i$, induces a snapshot B$_i$ (see Figure 3). In general, let $\mathbb{D}$ denote the first resolution of Di4; $\mathbb{D} := \{B_1, \dots B_b, \dots B_{|\mathbb{D}|}\}$ is the set of *snapshots* $B$ on $\Sigma$, as in Figure 3. By definition, the mapping $\mathbb{D} \rightarrowtail \Sigma$ is injective and non-surjective.

Di4 models a snapshot as a key-value pair element. The *key*, $e_b \in \Sigma$, is the *coordinate* of snapshot $B_b$ which refers to a location on the domain where a causal event has occurred; it is the unique identifier of $B_b$. The *value* is a tuple as $\langle \mu, \omega, \lambda \rangle$ (see Figure 3), where each component is defined as follows.

- The $\mu \in \mathbb{N}_0$ component is the count of non-causal events at the snapshot; e.g., see $\mu_2$ on Figure 3.
- The $\lambda$ component is the posting list of the snapshot, and it is a list of $\langle \varphi, @I \rangle$ tuples. Each tuple corresponds to a causal event, it references the event (using $@I$), and it informs whether the left ($\varphi := L$) or right ($\varphi := R$) end of the interval overlaps the snapshot key (e.g., see $\lambda_2$ and $\lambda_3$ on Figure 3). The $\varphi$ component has a retrieval optimization purpose: without it, Di4 should lookup a database by using the explicit reference to retrieve the interval coordinates and then compare these coordinates with the snapshot coordinate to determine overlaps; by using $\varphi$, the database lookup is avoided.
- The $\omega$ component is the number of causal intervals which overlap the snapshot with their right-end. In other words, the $\omega$ component is the count of posting list tuples with $\varphi = R$. This component also serves an optimization purpose. Using $\omega$, Di4 determines the number of intervals overlapping the snapshot with left and right ends in $O(1)$, respectively calculated as $|\lambda| - \omega$ and $\omega$. Otherwise, Di4 should linearly scan all tuples in the posting list. The number of intervals overlapping a snapshot with their left or right-end is used to calculate interval accumulation at a snapshot; this is a frequently used property in retrieval functions, and load-balanced partitioning for parallel processing.

#### 2.3.2 Indexing Algorithms

Di4 indexes intervals through a *batch indexing* procedure. In general, the procedure of indexing an interval requires two steps. First, it creates, or updates (if they already exist), two snapshots to bookmark the left and right ends of an interval, respectively $B_\alpha$ and $B_\gamma$. Second, it increments the $\mu_\beta, \alpha < \beta < \gamma$ component of $B_\beta$ snapshots.

A *single-pass* and a *double-pass* indexing algorithms have been defined. *Single-pass* indexing algorithm ensures consistency by correctly initializing $B_\alpha$, $B_\gamma$ and the $\mu_\beta$ components and maintains their value (see Algorithm 6 in appendix). *Double-pass* indexing neither fully initializes nor maintains $B_\alpha$, $B_\gamma$ and $\mu_\beta$ components at the *first-pass* (see Algorithm 7 in appendix), it rather ensures consistency only at the *second-pass* (see Algorithm 8 in appendix). The algorithms are explained using an example in appendix Section B.

The superiority of one algorithm over the other for indexing a sample $S$ depends on $|S|$ (number of intervals in the sample) and $|\mathbb{D}|$ (current number of snapshots in the first resolution). The single-pass indexing is optimal for updating Di4 data structure (i.e., when $|S| \ll |\mathbb{D}|$), while double-pass indexing is superior for initializing it (i.e., when $|S| \gg |\mathbb{D}|$, see appendix Section B).

#### 2.3.3 Interval Reconstruction

Di4 has an incremental structure, such that each snapshot has pointers to causal intervals only, and pointers to non-causal intervals are implied by neighbor snapshots (each snapshot is structured analogous to *P-frame* in video encoding). Pointers to the intervals are required to access metadata and execute UDFs; therefore, Di4 *reconstruct*s the intervals bookmarked by the snapshots to execute a query.

In general, given a snapshot, the *reconstruct* algorithm traverses its succeeding neighbor snapshots for the pointers to its non-causal intervals. The number of neighbor snapshots to be traversed, depends on the number of snapshots between the given snapshot, and the snapshot that bookmarks the right-most end of the non-causal events of the given snapshot. Therefore, given a snapshot, the number of neighbor snapshots to be traversed to reconstruct the bookmarked intervals, cannot be determined; it could be as small as 1, or as big as the size of whole first resolution. Therefore, the reconstruction process is potentially very expensive. However, Di4 significantly minimizes this number by a heuristic approach defined using $\lambda^*$ of secondary resolution; a process similar to the reconstruction of *P-frames* from an *I-frame* in video decoding (see Section 2.3.4). The pseudocode of reconstruction algorithm is given in Algorithm 4, and it is explained with an example in appendix Section C.

#### 2.3.4 Di4 Secondary Resolution

Di4 secondary resolutions are defined upon the first resolution indexing; they group snapshots and aggregate some attributes of the snapshots and/or bookmarked intervals. Different secondary resolutions can exist, being application-specific and independent from each other.

A secondary resolution is a set of *blocks* (see Figure 3), a block encapsulates a set of consecutive snapshots such that blocks do not have any snapshot in common. A block is a

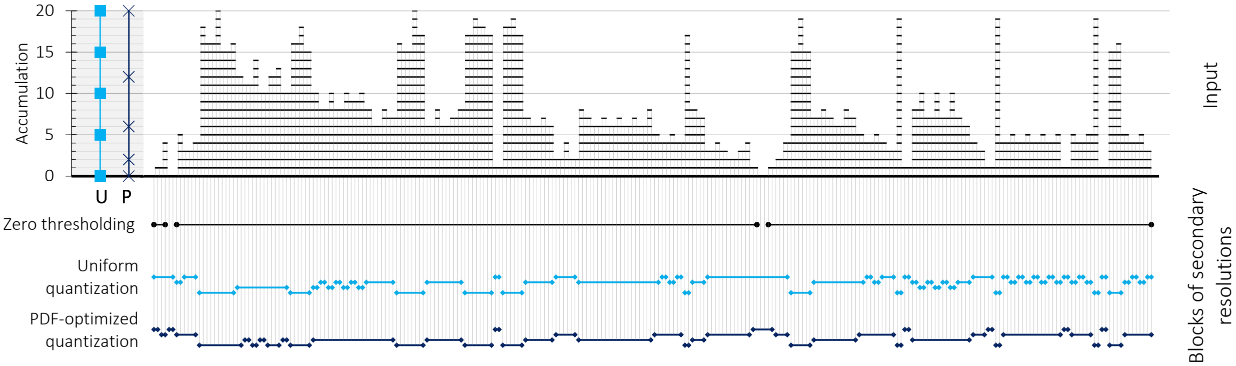


Fig. 4: Illustration of an example of creating blocks using the three built-in secondary resolution methods. **U**: uniform quantization boundaries, **P**: PDF-optimized quantization boundaries.

**Algorithm 4** The *reconstruct* algorithm consists of four procedures as explained here. The parameters $n$, $\lambda$, and $\lambda_{\text{tmp}}$ (initialized as $\lambda_{\text{tmp}} \leftarrow \emptyset$) are scoped to all the procedures of this algorithm.

```
procedure OPEN($B_b$, block)
  cache the block
  $n \leftarrow \mu_b - |\lambda_{\text{tmp}}|$        ▷ the number of snapshots to be reconstructed
  $\lambda \leftarrow \lambda_{\text{tmp}}$        ▷ the set of reconstructed intervals
  $\lambda_{\text{tmp}} \leftarrow \emptyset$
  INRECONSTRUCT($B_b$)
  for each interval in the last cached $\lambda$ whose right-end is not determined do
    add the interval to $\lambda$
    $n \leftarrow n - 1$

procedure INRECONSTRUCT($B_b$)        ▷ Inclusive Reconstruct
  for each interval $I$ bookmarked by $B_b$ do
    if the interval's left-end overlaps $B_b$ then
      add the interval to $\lambda$
    else
      UPDATELAMBDAS($I$)
  if the cached block's left-end overlaps $B_b$ then
    for each interval in $\lambda^*$ which is not in $\lambda$ do
      add the interval to $\lambda$ and all the cached $\lambda$s
      $n \leftarrow n - 1$

procedure EXRECONSTRUCT($B_b$, $U$, Args)        ▷ Exclusive Reconstruct
  for each interval $I$ bookmarked by $B_b$ do
    if the interval's left-end overlaps $B_b$ then
      add the interval to $\lambda_{\text{tmp}}$
    else if the interval $I$ is in $\lambda_{\text{tmp}}$ then
      remove the interval from $\lambda_{\text{tmp}}$
    else
      UPDATELAMBDAS($I$)
  if $n = 0$ then
    for each ⟨Args, $\lambda$⟩ do        ▷ including cached Args and $\lambda$s
      U(Args, $\lambda$)
    return true
  else
    return false

procedure UPDATELAMBDAS($I$)
  if the interval $I$ is not in the last cached $\lambda$ then
    add the interval $I$ to all the $\lambda$s
    $n \leftarrow n - 1$
  else
    set its $\varphi = R$ in all the $\lambda$s        ▷ i.e., the interval's right-end is determined
```

key-value pair element where the *key* is the first and last point on the domain that are bookmarked by the encapsulated snapshots. The key is defined using a user-defined *grouping function*; its application is described in Algorithm 5. Di4 has three built-in grouping functions defined in Section 2.3.5.

The *value* has two parts; the first part is a user-defined tuple of aggregated attributes of encapsulated snapshots and/or intervals. The second part ($\lambda^*$) is a list of pointers to the non-causal intervals overlapping the left-most encapsulated snapshot; such that all intervals overlapping this snapshot are reconstructed independent from neighbor snapshots (this property makes the snapshot analogous to *I-frame* in video encoding).

**Algorithm 5** Indexing second resolution. $\Theta$ is a secondary resolution partitioning function, it could use any of the default functions (see Section 2.3.5) or a user-defined function. The UDF ($U$) aggregates attributes of bookmarked intervals or snapshots as *value* of secondary resolution blocks.

```
procedure SECONDARYRESOLUTION($\Theta$, $U$)
  $a \leftarrow 0$
  $t \leftarrow \{\}$
  $\lambda^* \leftarrow \{\}$
  initialize $\Theta$ with accumulation at $B_0$, and $\lambda_0$
  for each snapshot $B_b$ do
    if $\Theta$(accumulation at $B_b$, $\lambda_b$) = true then
      insert a new block to secondary resolution initialized as:
        key: $[e_a, e_b]$, value: $\langle \lambda^*, U(t) \rangle$
      $a \leftarrow b$
      $t \leftarrow \lambda^*$
      insert all intervals starting at $B_b$ to $t$
    else
      insert all intervals in $\lambda_b$ to $t$
    insert all intervals starting at $B_b$ to $\lambda^*$
    remove all intervals stopping at $B_b$ from $\lambda^*$
```

The motivations of secondary resolutions are threefold; first, a heuristic approach for pruning the number of snapshots to be processed for executing a query without altering the design of the underlying data structure, thereby optimizing the performance of specific queries. For instance, let us consider a computational biology application where Di4 is commonly queried with *coordinate* (indexed attribute) and *statistical significance* (*p-value*, an application specific non-indexed attribute) criteria. Without a secondary resolution, Di4 finds all the *candidate* intervals that comply *coordinate* criteria, and then passes them to a UDF to be filtered by *p-value*, and possibly further processed. However, with a secondary resolution which groups consecutive snapshots by *p-value*, Di4 can search *candidate* intervals that comply *coordinate* criteria only in the groups that comply the *p-value* criterion, which minimizes the number of *candidates* to be passed to a UDF for possible further processing.

Second, secondary resolutions are used to optimize parallel execution. In an application with Di4 being used in a Cloud environment over Big data, where it is essential to optimally distribute workloads across multiple computing resources, secondary resolution can efficiently split data into *load-balanced* partitions (bins), and then allocating partitions evenly across all nodes. This is a load-balancing policy which minimizes the idle time of computing resources.

Third, secondary resolution is finally used to optimize the reconstruction of bookmarked intervals. Indeed, Di4 leverages $\lambda^*$ of the closest *block* to reconstruct the intervals bookmarked by a snapshot. With a balanced secondary resolution (see Section 2.3.5, this significantly reduces the number of snapshots to be traversed in a reconstruction process, hence increasing the reconstruction speed, and accordingly, the query execution time.

A secondary resolution is not equivalent to a secondary index [28], [29]. A secondary resolution index is commonly defined on the same attribute as the primary resolution, while primary and secondary indexes are commonly defined on different attributes. For instance, while a primary resolution of Di4 indexes coordinates of intervals, its secondary resolution can index groups of snapshots bookmarking position on the domain overlapping various functional portions of the genome (e.g., gene body, or transcription factor binding site).

#### 2.3.5 Default Secondary Resolutions

Di4 implements 3 default methods to create a secondary-resolution, listed below in increasing order of complexity: (1) Zero thresholding, (2) Uniform scalar quantization (SQ) and (3) probability density function (PDF) optimized scalar quantization (where (2) and (3) are two variants of scalar quantization). In the following, we describe each of these methods:

1) *Zero thresholding*, it defines a block as a set of contiguous snapshots all bookmarking at least one interval (see Figure 4).
2) *Uniform scalar quantization*. The goal of quantization is to approximate a distribution of given points with $2^n$ points, where $n$ is the number of quantization levels. A scalar quantization is a function that maps its input to distinct regions (quantization regions), and represents each region by a point (reconstruction point). Here the scalar quantization method is defined over accumulation of intervals; and it defines a block as a set of contiguous snapshots all belonging to the same reconstruction point, i.e., it breaks a block at a snapshot with different reconstruction points with respect to its prior snapshot. In uniform quantization, the quantization regions are equally spaced, and the reconstruction levels are at the midpoint of each interval.
3) *PDF-optimized scalar quantization*. In this modified quantization scheme, the quantization regions are shortened or lengthened according to the probability of each region. We adopted the well-known Lloyd-Max quantization for our purpose [30]. In this method, the quantization reconstruction levels are the centroid, or center of mass, of the signal PDF in the related quantization regions.

We provide an example of creating blocks using the three built-in secondary resolution methods as illustrated in Figure 4. The upper part of the figure shows synthetic input intervals, where for each position on the domain a snapshot is created; however, for readability of the figure, the snapshots are not displayed. The lower part of the figure shows how consecutive snapshots are organized in blocks using the three built-in methods, where each level of lines represent a secondary resolution method, and line breaks at each level represent different blocks. The quantization regions are shown on the left-most to the input, vertical lines above $U$ and $P$. As it can be seen, the quantization regions have equal distances in uniform quantization and variable/unequal distances in PDF-optimized approach.

By looking at the figure one can note how differently consecutive snapshots are organized in blocks using three methods, *quantitatively* (i.e., zero-thresholding *v.s.* scalar quantization methods) or *qualitatively* (i.e., uniform *v.s.* PDF-optimized scalar quantization).

### 2.4 Di4 Data Serialization

The Di4 serialization process (de)serializes a Di4 snapshot into an array of bits, then the persistence technology organizes the array in its internal structure. The Di4 design is agnostic to a key-value pair persistence technology (see Figure 1), hence Di4 does not implement how a serialized snapshot is organized and persisted on disk. This design allows us to focus on an optimal (de)serialization of a snapshot independent from its organization on disk. Di4 leverages serialization methods used in *protocol buffers* [31], and serializes a snapshot into an "arranged" binary representation, which uses fewer bits than common serialization methods (e.g., JavaScript Object Notation) to serialize an object. Additionally, Di4 uses the *variable-length quantity* method to encode an unsigned integer in a compact representation, commonly referred-to as *7-bit encoded int* or *varint* [31], [32]. Accordingly, Di4 concisely serializes a snapshot. For instance, it serializes the $B_2$ snapshot in the Figure 3 using 48 bits, which would require at least 136 bits otherwise. Detailed discussion is available in Section E of appendix.

## 3 Experimental Evaluation

The present section provides a benchmark of Di4, and a comparison with the state of the art.

### 3.1 Experimental and Environment Setup

Di4 is customized for genomics with Di4B (Di4 for Bioinformatics) at business logic layer, and Di4BCLI (Di4B Command Line Interface) at presentation layer. Di4B defines a genomics-specific environment setup (e.g., define the domain), and initializes several independent Di4 instances, one for each DNA chromosome and strand. Di4BCLI is a command-line interface which provides user interaction through a set of commands which has been used in the experiments (see Figure 1).

The performance of Di4 is evaluated using samples downloaded from ENCODE which is a public repository of NGS data. The downloaded data are grouped in 9 datasets as described in Table 3, the A4 dataset is the current biggest publicly available dataset from this public repository. See appendix Section D for details on the datasets.

Performance is assessed on a current modern machine with specifications summarized on Table 2. Theoretical peak performance of the machine's processor is given in Giga Floating Point Operations Per Second (GFLOPS). The machine has a Solid-State Drive (SSD) storage device, which is assessed for sequential read/write (i.e., the time it takes to read and write a 1GB file), and random read/write of 4K blocks.

Di4 runs at a user-defined degree-of-parallelism (dp), defined as *Di4B-level dp* $\times$ *Di4-level dp*, that is respectively the number of independent instances of Di4 (i.e., chromosomes and strands) being executed concurrently, and the number of threads read/write each Di4 instance. The experiments have been performed using $4 \times 2 = 8$ threads. The tools used later in this section to benchmark Di4, are run in their native degree of parallelism (i.e., we do not implement a parallelization method if they run single-threaded, or modify their parallelization capabilities). The scripts and data used for running benchmarks presented in this section are available at genometric.github.io/Di4/benchmark.

Note that Di4 is defined at data access layer, and it does not implement a persistence technology (see Figure 1). Therefore, its performance can vary depending on the technology utilized at persistence level; to minimize the bias on persistence technology, we benchmark using a B+tree implementation (github.com/csharptest/CSharpTest.Net.Collections).

### 3.2 Di4 Operations Benchmark

The `INTERSECT` and `COVERAGE` functions are fundamental to Di4 operations; accordingly, their performance is in direct relation with the performance of the majority of Di4 operations. Therefore, these functions are primarily benchmarked.

`INTERSECT`: "How long it takes Di4 to find all intervals overlapping a reference interval?". This operation includes finding a pivot snapshot (i.e., the snapshot that overlaps or is the closest down-stream to the left-end of a reference interval), traversing snapshots, and reconstructing the bookmarked intervals (see Algorithm 1). This operation is benchmarked using datasets A1-A4, and the results are plotted on panel A of Figure 6. The results are based on $10$ executions of `INTERSECT` function on $196,180$ reference intervals (i.e., $1,961,800$ runs of `INTERSECT` function). The query processing time does not include data indexing time.

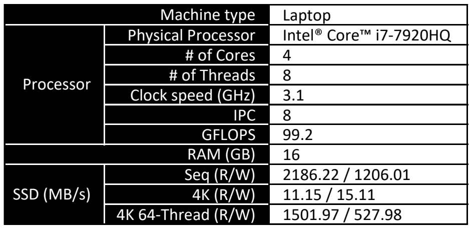
`COVERAGE`: "How long it takes Di4 to assess the compliance of all snapshots with a given coverage function?". This operation includes traversing the snapshots, reconstructing the bookmarked intervals, and check for the compliance with the given coverage function. Here we benchmark Di4 for *coverage of accumulation* (i.e., `COVER` function), with and without the utilization of a secondary resolution (created using PDF-optimized scalar quantization) using datasets A1-A4. We benchmarked using $20$ accumulation ranges (same query ranges as in appendix section A) and executed each range for $10$ times (i.e., $200$ executions of `COVER` function). The results are plotted on panels B and C of Figure 6 as snapshot bulk processing speed (i.e., snapshot per second). The query processing time does not include data indexing time.

### 3.3 Inverted vs. Incremental Inverted Index

A previous indexing framework for genomic intervals, which we have taken inspiration from, is called Di3 [4]. Di3 and Di4 have a common goal of providing the genomics data processing with a holistic and extensible information retrieval framework, but they have fundamental differences in the model, and Di4 benefits from a significantly more effective secondary resolution methods. As a result, Di4 executes indexing and retrieval functions significantly faster than Di3, while having a considerably smaller index size. The differences are discussed in details in the following.

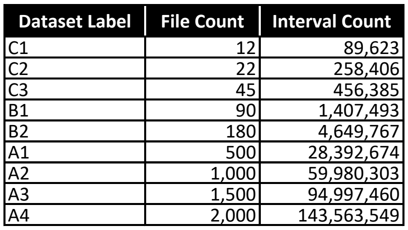
The fundamental design decision making the difference between Di4 and Di3 is at model level; while Di3 leverages the inverted index paradigm, Di4 has an incremental inverted index structure, yielding to different first resolution indexes. In general, for each position on the domain, Di3 bookmarks *causal* and *non-causal* intervals, while Di4 bookmarks only *causal* intervals (see Section 2.1). This design makes $\lambda$ component (posting list) of Di4 snapshots significantly smaller than the $\lambda$ component of Di3 snapshots; our test using the A4 dataset shows $4\times$ smaller components (see panel A on Figure 5), an immediate effect of which is the (approximately) $5\times$ smaller index file size (see panel B on Figure 5), without penalizing indexing operation (see panel C on Figure 5). A smaller snapshot is faster to deserialize and process, hence making Di4 operations significantly faster than Di3 operations; for instance, testing `COVERAGE` function shows 2–$12\times$ expedited runtime (see panel E on Figure 5). Note that `COVERAGE` is the base function of most Di4 operations, hence similar expedited runtime is expected from all the functions which stem from `COVERAGE`.

However, the expedited runtime is only partially due to the smaller snapshots, and its partially due to the heuristically more efficient secondary resolution blocks (see Section 2.3.4), which (a) heuristically decreases the number of snapshots to be processed when executing a query, and



| | | |
|---|---|---|
| Machine type | | Laptop |
| Processor | Physical Processor | Intel® Core™ i7-7920HQ |
| | # of Cores | 4 |
| | # of Threads | 8 |
| | Clock speed (GHz) | 3.1 |
| | IPC | 8 |
| | GFLOPS | 99.2 |
| RAM (GB) | | 16 |
| SSD (MB/s) | Seq (R/W) | 2186.22 / 1206.01 |
| | 4K (R/W) | 11.15 / 15.11 |
| | 4K 64-Thread (R/W) | 1501.97 / 527.98 |

TABLE 2: The specifications of the machine used for benchmarking.



| Dataset Label | File Count | Interval Count |
|---|---|---|
| C1 | 12 | 89,623 |
| C2 | 22 | 258,406 |
| C3 | 45 | 456,385 |
| B1 | 90 | 1,407,493 |
| B2 | 180 | 4,649,767 |
| A1 | 500 | 28,392,674 |
| A2 | 1,000 | 59,980,303 |
| A3 | 1,500 | 94,997,460 |
| A4 | 2,000 | 143,563,549 |

TABLE 3: The datasets used for benchmarks, which are downloaded from ENCODE.

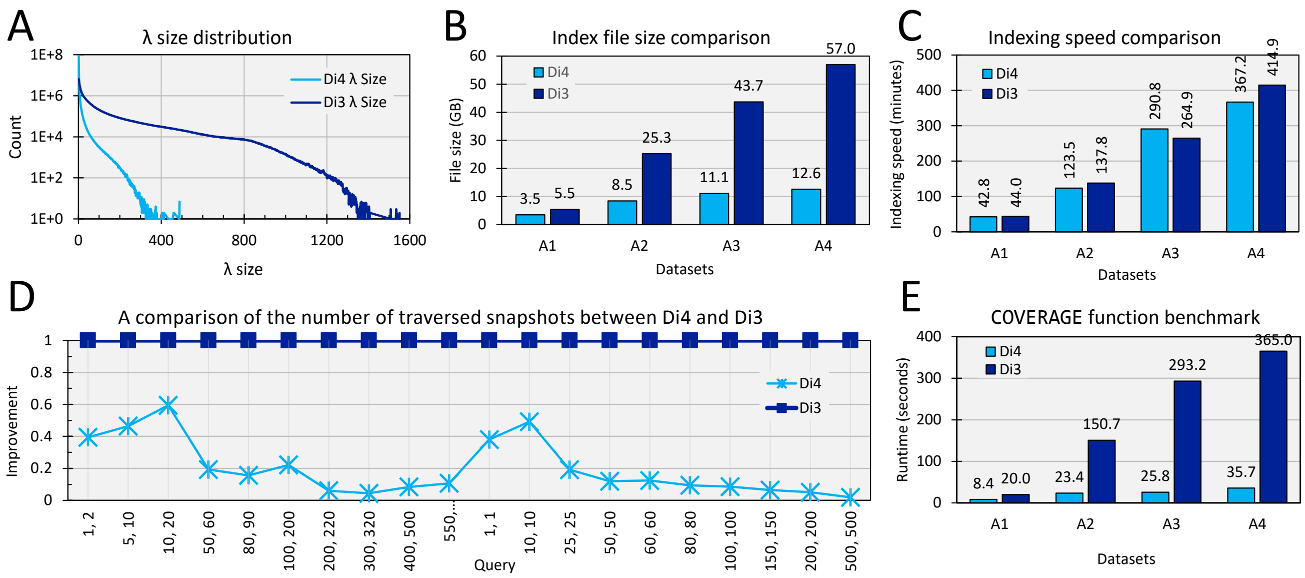


Fig. 5: Panel **A** compares the $\lambda$ size of Di3 and Di4 when indexing the A4 dataset; it shows that Di4 has significantly smaller $\lambda$ component which results into a considerably smaller index file size, shown in panel **B**. Panel **C** shows Di4 indexing speed is comparable to Di3 (both running a double-pass indexing method). The secondary resolutions of Di4 heuristically minimize the number of snapshots to be traversed when executing a query, panel **D** plots this improvement *w.r.t.* Di3. As a result of smaller snapshots (panel **A**), smaller index size (panel **B**), and heuristically improved query execution (panel **D**), Di4 runs significantly faster than Di3 (panel **E**).

(b) optimally load-balances parallelization. Our test using the A4 dataset shows that Di4 executes the same query as Di3 traversing 10–60% (20% on average) of the snapshots traversed by Di3 (see panel D on Figure 5).

Having bookmarked all intervals overlapping a position on domain in a Di3 snapshot, Di3 can "find all intervals overlapping a point on domain" (queries similar to segment trees) leveraging a single snapshot; while executing such queries on Di4 would require traversing *at least one snapshot* to reconstruct all the intervals overlapping the given point. However, our tests for "find all intervals from the A4 dataset overlapping 196, 180 reference intervals" (where the length of each reference interval is 1) using both Di3 and Di4, shows that Di4 runs faster than Di3 (12.75sec and 137.17sec for Di4 and Di3 respectively as the average of 10 executions). This observation emphasizes that traversing Di3's bigger index (*w.r.t.* Di4) and deserializing a single snapshot, is slower than traversing Di4's smaller index and deserializing at least one smaller snapshot (*w.r.t.* Di3).

### 3.4 Comparison with BEDTools and BEDOPS

In this section, the performance of Di4 has been benchmarked against Di3 [4] and current latest versions of two commonly used tools in bioinformatics, BEDTools [13] (version 2.27.1) and BEDOPS [14] (version 2.4.32). Given that BEDTools and BEDOPS run on two input samples, scripts for their batch execution have been prepared (available at genometric.github.io/Di4/benchmark). Di4 `INTERSECT` has been benchmarked against `bedtools intersect`, `bedops intersect`, and `MAP` from Di3. The performance is evaluated in three scenarios, covering typical *dry-lab* experiments, discussed as follows.

On-the-Fly Processing: The daily-based data processing activity of a bioinformatician is running a NGS data processing pipeline, obtaining a relatively small dataset, and evaluating comparatively this dataset with related small datasets. Based on the results, the outcome should either be archived for further processing, or discarded. Di4 is benchmarked against Di3, BEDTools and BEDOPS for this scenario on the `INTERSECT` operation using a reference sample from "ENCODE narrow peak" repository which contains 196, 180 intervals, and the C1, C2, and C3 target dataset (see Table 3). Additionally, since BEDTools and BEDOPS run in memory, Di3 and Di4 are also executed in memory. This on-the-fly processing scenario consists of processing and pre-processing; therefore, the query runtime incorporates pre-processing, which is sorting data for BEDTools and BEDOPS, and indexing for Di4 and Di3. The results, which are the average of 10 executions, are plotted on panel D of Figure 6 as total query runtime.

Personal Repository: This is also a common scenario for bioinformaticians, where a personal repository of in-house data is comparatively evaluated or cross-referenced for further assessments. Di4 is benchmarked against Di3, BEDTools and BEDOPS for this scenario on `INTERSECT` operation using B1, and B2 datasets (see Table 3) and a reference sample from "ENCODE narrow peak" repository which contains 196, 180 intervals. Since BEDTools and BEDOPS run in-memory, Di4 and Di3 are also executed in-memory. Given that a personal repository is a collection of properly organized data (i.e., sorted, concerning BEDTools and BEDOPS, or indexed concerning Di4 and Di3), the query time excludes pre-processing time in all cases. The results, which are the average of 10 executions, are plotted on panel E of Figure 6 as total query runtime.

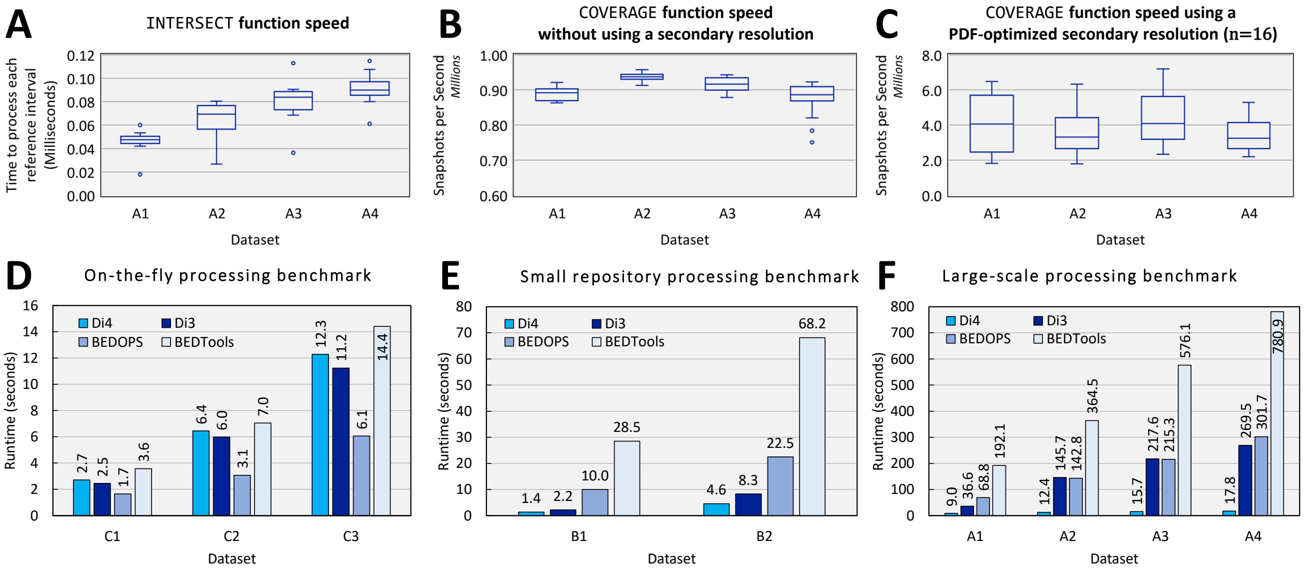


Fig. 6: Benchmarking Di4 operations. Panels **A**, **B**, and **C** plot the performance of Di4 base operations. Panels **D**, **E**, and **F** plots the performance of Di4 against Di3, BEDTools, and BEDOPS (each runtime is the average of 10 executions).

Large-Scale Scenario: As public repositories of NGS data are rapidly growing, *sense-making* from NGS data through large-scale comparative evaluation is becoming ubiquitous. This highlights a demand for holistic and scalable framework for comparative evaluation of NGS data. Di4, Di3, BEDTools, BEDOPS are benchmarked for this scenario using datasets A1-A4. Since such repositories are collection of properly organized and persisted data (i.e., sorted, concerning BEDTools and BEDOPS, or indexed concerning Di4 and Di3), the query time excludes pre-processing time. Di4 and Di3 are both executed using a persisted index. These tools are benchmarked as average of 10 executions, and the results are plotted on panel F of Figure 6.

## 3.5 Comparison with Giggle

Giggle [16] is a tool for querying genomic datasets, which leverages an index structure similar to Di3. Giggle finds indexed intervals overlapping a given set of query intervals, and ranks the results using the product of $-\log_{10}$(p-value) and $\log_2$(odds ratio). Di4 and Giggle can be compared from two facets; first, unlike Giggle that is defined for a particular application scenario, Di4 is a framework implementing "building blocks" for a wide-variety of application scenarios. Accordingly, while Giggle mainly targets an end-user, Di4 is designed for developers who can augment it for their particular application.

Second, the performance of Di4's most similar functionality to Giggle. Accordingly, Di4 is benchmarked against Giggle (version 0.6.3) for querying from small and large datasets, respectively, Roadmap Epigenomics dataset (used in [16] for benchmarking Giggle) containing $1,905$ samples and $55,558,166$ intervals, and A4. The datasets are queried using 5 samples downloaded from ENCODE (see appendix Table 4) with a varying number of intervals in each, spanning from $29,972$ to $442,035$. Each query sample is queried 10 times using Di4 and Giggle, and their average runtime is plotted in Figure 7. As plotted in Figure 7, Di4 runs up to 6× faster than Giggle.

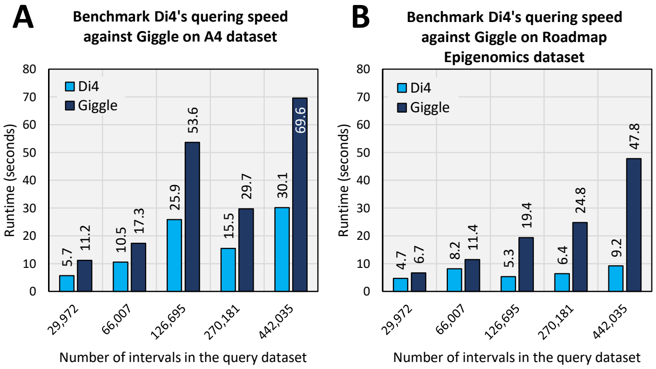


Fig. 7: The runtime plotted here is the average of 10 executions of each query. The results show that Di4 runs up to 3× faster than Giggle on a large dataset (panel A), and up to 6× faster on a small dataset (panel B).

## 3.6 Evaluation of second resolution

One of the main goals of secondary resolutions is to optimize querying non-sargable attributes. Without utilizing a secondary resolution, executing such queries would require linearly scanning the entire first resolution. Therefore, in this section we benchmark the default secondary resolutions on how much they expedite executing such queries.

As explained in Section 2.2, *Coverage* on $f$ is one of the most common queries on non-indexed attributes in genomic research. Accordingly, we define a *base* query, which is the execution time of the `COVER` function without a secondary resolution (i.e., linearly scanning the entire first resolution).

We evaluated the performance of the different secondary resolution indexing methods using A4 dataset (see Table 3). The query execution time is used to assess the performance of the various methods and the results were normalized *w.r.t.* the *base* query execution time to show the improvement with respect to the latter. The base query time was measured as $t_{\text{base}} = 177.89$ sec, which is the average of 10 executions. A normalized query time of smaller than 1 shows improvement in querying execution time.

The panels A and B on Figure 8 show the query times for the range and point queries with respect to the defined query ranges. As it can be seen on both panels, all the 3 curves lie below the base query time (i.e., the horizontal line at 1). This is an important observation and highlights the effectiveness of the proposed secondary resolution methods in improving the retrieval performance.

Our results also indicate that the PDF-optimized scalar quantization method provides the best performance compared with the other two secondary resolution schemes where the amount of improvement is approximately 80% on average. We carried out 1-way ANOVA to investigate whether there is a significant difference between the means of three secondary resolution methods [33] (i.e., the means of the curves shown on Figures 7). The statistical test reveals that the PDF-optimized quantization methods outperforms other approaches with statistically significant difference ($p < 0.05$ i.e., with confidence level of 95%) while the other two approaches do not show a significant difference with respect to each other. This is an interesting outcome and can highlight the promising results that can be achieved if the proposed quantization scheme is utilized to create the secondary resolutions. Indeed:

- base query linearly scans the entire first resolution, which, utilizing a secondary resolution, linear scan is limited to particular sets of consecutive snapshots (i.e., the snapshots encapsulated by a block whose aggregated attribute overlaps the query criteria).
- the reconstruction time of intervals bookmarked by snapshots is expedited using the $\lambda^*$ component of blocks which allows reconstruction of intervals independent from neighbor snapshots (similar to keyframes in video encoding/decoding).
- an optimal parallelization would require independent regions for each thread/node; however, splitting first resolution to independent regions without scanning it, would be a challenge. Each block of secondary resolution defines a set of snapshots which can be processed independently and in parallel with other snapshots. Zero-thresholding defines the simplest splitting, and PDF-optimized defines load-balanced regions.

## 4 Conclusion and Future Work

Di4 is an instrument for fast indexing of large repositories for tertiary data analysis, supporting very fast interval-based operations over region-based heterogeneous genomic datasets; in comparison with other interval-based data management systems, it supports abstractions that make it the most suitable tool for *making sense* of genomic data. We expect this application to became increasingly important, as the availability of processed genomic datasets is growing at an huge and unprecedented speed; genomic data integration will be key to major discoveries in biology and will open the route to personalized medicine.

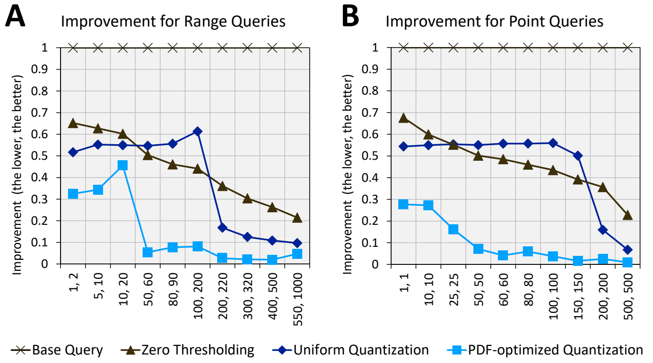


Fig. 8: Second resolution evaluations. **A**, **B** the performance of the three built-in methods for secondary resolution normalized to base query time.

Di4 is a single-dimensional index, and we envision a future work toward a multi-dimensional index paradigm for two primary reasons; first, improve performance for executing particular location-agnostic queries. For instance, while leveraging secondary resolutions, Di4 can perform a heuristically-optimized liner scan to "find regions where at least any $n$ samples overlap", it cannot leverage the same heuristics to "find regions where at least $n$ of the given samples overlap". An optimal execution of the latter query demands an additional dimension to Di4 where intervals are indexed based on a sample ID to which they belong.

Second, genome is commonly modeled linearly; however, recent advances unfold long-range interactions that can be explained considering spatial organization of DNA. To adapt with this emerging trend, new coordinate attributes should be identified and incorporated into Di4's model.

## Acknowledgments

This research is funded by the European Research Center (ERC) (Advanced ERC Grant 693174) Project "Data-Driven Genomic Computing (GeCo)".

## References

[1] J. Shendure, S. Balasubramanian, G. M. Church, W. Gilbert, J. Rogers, J. A. Schloss, and R. H. Waterston, "Dna sequencing at 40: past, present and future," *Nature*, vol. 550, no. 7676, pp. 345–353, 2017.

[2] Y. Kodama, M. Shumway, and R. Leinonen, "The sequence read archive: explosive growth of sequencing data," *Nucleic acids research*, vol. 40, no. D1, pp. D54–D56, 2011.

[3] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS biology*, vol. 13, no. 7, p. e1002195, 2015.

[4] V. Jalili, M. Matteucci, M. Masseroli, and S. Ceri, "Indexing next-generation sequencing data," *Information Sciences*, 2016.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Section 14.3: Interval trees*, third edition ed. MIT press Cambridge, ch. 14, pp. 348–354.

[6] J. L. Bentley, "Solutions to klees rectangle problems," Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1977.

[7] ——, "Decomposable searching problems," *Information processing letters*, vol. 8, no. 5, pp. 244–251, 1979.

[8] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.

[9] A. Guttman, *R-trees: a dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.

[10] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, “The tiledb array data storage manager,” *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.

[11] R. M. Layer, K. Skadron, G. Robins, I. M. Hall, and A. R. Quinlan, “Binary interval search: a scalable algorithm for counting interval intersections,” *Bioinformatics*, vol. 29, no. 1, pp. 1–7, 2012.

[12] U. Paila, B. A. Chapman, R. Kirchner, and A. R. Quinlan, “Gemini: integrative exploration of genetic variation and genome annotations,” *PLoS computational biology*, vol. 9, no. 7, p. e1003153, 2013.

[13] A. R. Quinlan and I. M. Hall, “Bedtools: a flexible suite of utilities for comparing genomic features,” *Bioinformatics*, vol. 26, no. 6, pp. 841–842, 2010.

[14] S. Neph, M. S. Kuehn, A. P. Reynolds, E. Haugen, R. E. Thurman, A. K. Johnson, E. Rynes, M. T. Maurano, J. Vierstra, S. Thomas *et al.*, “Bedops: high-performance genomic feature operations,” *Bioinformatics*, vol. 28, no. 14, pp. 1919–1920, 2012.

[15] E. Afgan, D. Baker, B. Batut, M. van den Beek, D. Bouvier, M. Čech, J. Chilton, D. Clements, N. Coraor, B. A. Grüning *et al.*, “The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update,” *Nucleic acids research*, vol. 46, no. W1, pp. W537–W544, 2018.

[16] R. M. Layer, B. S. Pedersen, T. DiSera, G. T. Marth, J. Gertz, and A. R. Quinlan, “Giggle: a search engine for large-scale integrated genome analysis,” *Nature methods*, 2018.

[17] M. K. Aguilera, W. Golab, and M. A. Shah, “A practical scalable distributed b-tree,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[18] N. Dayan, M. Athanassoulis, and S. Idreos, “Monkey: Optimal navigable key-value store,” in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 79–94.

[19] J. F. Roddick and M. Spiliopoulou, “A survey of temporal knowledge discovery paradigms and methods,” *IEEE Transactions on Knowledge and data engineering*, vol. 14, no. 4, pp. 750–767, 2002.

[20] G. Ozsoyoglu and R. T. Snodgrass, “Temporal and real-time databases: A survey,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 513–532, 1995.

[21] C. S. Jensen, C. E. Dyreson, M. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer *et al.*, “The consensus glossary of temporal database conceptsfebruary 1998 version,” in *Temporal Databases: Research and Practice*. Springer, 1998, pp. 367–405.

[22] V. J. Tsotras and N. Kangelaris, “The snapshot index: an i/o-optimal access method for timeslice queries,” *Information Systems*, vol. 20, no. 3, pp. 237–260, 1995.

[23] R. Elmasri, G. T. Wuu, and Y.-J. Kim, “The time index: An access structure for temporal data,” in *Proceedings of the 16th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1990, pp. 1–12.

[24] C. H. Goh, H. Lu, B.-C. Ooi, and K.-L. Tan, “Indexing temporal data using existing b+-trees,” *Data & Knowledge Engineering*, vol. 18, no. 2, pp. 147–165, 1996.

[25] V. Jalili, M. Matteucci, M. J. Morelli, and M. Masseroli, “Musera: multiple sample enriched region assessment,” *Briefings in bioinformatics*, vol. 18, no. 3, pp. 367–381, 2016.

[26] V. Jalili, M. Matteucci, M. Masseroli, and M. J. Morelli, “Using combined evidence from replicates to evaluate chip-seq peaks,” *Bioinformatics*, vol. 31, no. 17, pp. 2761–2769, 2015.

[27] N. F. Wasserman, I. Aneas, and M. A. Nobrega, “An 8q24 gene desert variant associated with prostate cancer risk confers differential in vivo activity to a myc enhancer,” *Genome research*, vol. 20, no. 9, pp. 1191–1197, 2010.

[28] T. Kahveci and A. Singh, “Variable length queries for time series data,” in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 273–282.

[29] S. Kadiyala and N. Shiri, “A compact multi-resolution index for variable length queries in time series databases,” *Knowledge and Information Systems*, vol. 15, no. 2, pp. 131–147, 2008.

[30] S. P. Lloyd, “Least squares quantization in pcm,” *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, 1982.

[31] Encoding protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding#varints

[32] Binarywriter. [Online]. Available: http://referencesource.microsoft.com/#mscorlib/system/io/binarywriter.cs

[33] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

**Vahid Jalili** was awarded his Ph.D. degree (cum laude) on Information Technology at the Politecnico di Milano, Italy, in 2016. His research in biomedical computing area spans novel systematic solutions for analytical and computational challenges. He is a team member of the Galaxy project (http://galaxyproject.org), a biomedical data analysis platform. His research interest spans topics in computational biology, cognitive science, and artificial intelligence.

**Matteo Matteucci** is Associate Professor at the Dipartimento di Elettronica Informazione e Bioingegneria of Politecnico di Milano. His main research topics are pattern recognition, machine learning, machine perception, and signal processing. His main research interest is in developing, evaluating and applying, in a practical way, techniques for adaptation and learning to real world autonomous systems.

**Jeremy Goecks** is Assistant Professor of Biomedical Engineering and Computational Biology at Oregon Health and Science University (OHSU). His research is focused on analysis methods for genomics and biomedical imaging, visual analytics tools for large datasets, machine learning for precision medicine, and computational infrastructure for biomedical data science.

**Yashar Deldjoo** is a Phd Student in Computer Science at the Department of Electronics, Information and Bioengineering (DEIB) of Politecnico di Milano, Italy. His main research topics include machine learning, recommender systems and signal processing. As a young researcher, he has published several articles papers in different peer-reviewed journals and conferencs.

**Stefano Ceri** is Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. His research has been generally concerned with extending database technology; he has authored over 300 publications (H-index 74) and 15 books in English. He received two advanced ERC Grants, on Search Computing and on Data-Driven Genomic Computing (GeCo, 2016-2021). He received the ACM-SIGMOD Innovation Award (2013) and is an ACM Fellow.