Fast Query and Interactively Explore Data from Indexed Genomic Files

Yifan Yang
yang7832@umd.edu
Center for Bioinformatics and Computational Biology
Department of Computer Science
College Park, Maryland

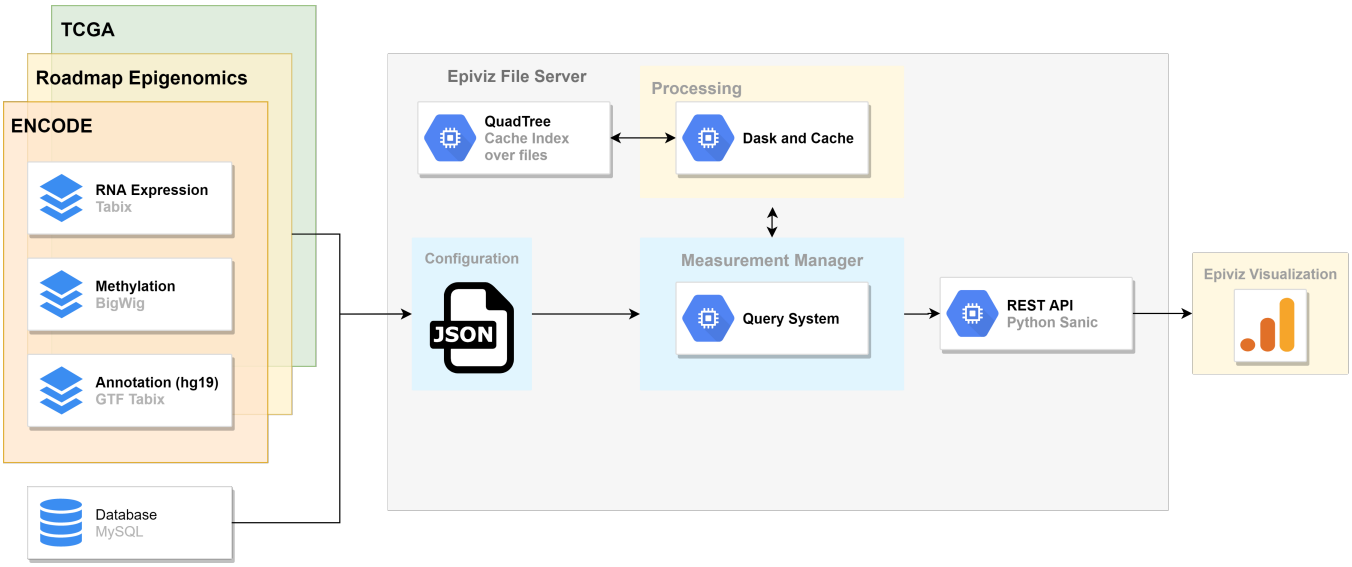

Figure 1: A high level overview of the Epiviz File Server [10] Library. Epiviz File Server library supports directly querying indexed genomic files. Data files are described in the measurements module and provides a programmatic interface to parse, query and define transformations over files indexed by Quadtree using any NumPy(-like) function. Transformations are lazily computed at query time using Dask and the cache layer makes sure we only request for bytes not already accessed. Datasets and their transformations can be accessed using a REST API and allows developers to build interactive visualization and exploration tools.

ABSTRACT

Genomic data repositories like The Cancer Genome Atlas (TCGA), Encyclopedia of DNA Elements (ENCODE), Bioconductor's AnnotationHub and ExperimentHub etc., provide public access to large amounts of genomic data as flat files. Researchers often download a subset of data files from these repositories to perform exploratory data analysis. We developed Epiviz File Server, a Python library that implements an in-situ data query system for local or remotely hosted indexed genomic files, not only for visualization but also data transformation. The File Server library decouples data retrieval and transformation from specific visualization and analysis tools and provides an abstract interface to define computations independent of the location, format or structure of the file. We also present new approaches to quickly query genomic files using space-partition indexing data structures. This index can be used for any interval datasets but in our approach we use this in the context of genomic data.

1 MOTIVATION AND PREVIOUS WORK

Genomic data repositories like The Cancer Genome Atlas [6], Encyclopedia of DNA Elements [5] (ENCODE), Bioconductor's [9] AnnotationHub [17] and ExperimentHub [15] etc., provide public access to large amounts of genomic data as flat files. Researchers often download a subset of files data from these repositories to perform their data analysis. As these data repositories become larger, researchers often face bottlenecks in their exploratory data analysis. Increasing data size requires longer time to download, pre-process and load files into a database, to run queries efficiently.

Interactive visualization of data can be a powerful tool to enable exploratory analysis. As users get familiar with the data and gain insights, it would be even more efficient to interactively hypothesize, validate, visualize and compute the intermediate results of the analysis. Currently available interactive visualization tools for genomic data, namely genome browsers such as Epiviz [4] [11], fall into two broad categories. One that uses a database management system to load genomic data from files into tables, create indices or partitions for faster query of data by genomic intervals. As genomic

data can be huge, importing large data into databases can take a significant amount of time thus making this process often unfeasible. The other category of genome browsers query data directly from indexed genomic file formats like BigBed, BigWig [12] or Tabix [14]. However these tools are limited only to exploration of data from files.

Genomic files hosted on public repositories are fairly stable and often do not change. Also when the library queries the file for data, it first parses the index of these files. Given these conditions, we can either pre-merge the index of all files or use a data structure to dynamically update indices as we parse files. This would be similar to the interval data structure used by GIGGLE [13], which uses a B+ tree to create an index from thousands of genomic data and annotation files.

Our previous approach in EFS uses indices for individual genomic files to query for data. The query efficiency for this system does not scale well with large numbers of files. Here, we present an extension to EFS that indices multiple files in a single data structure. We also present new approaches to quickly query genomic files using space-partition indexing data structures. This index can be used for any interval datasets but in our approach we use this in the context of genomic data.

2 METHODS

2.1 Epiviz File Server

Based on the concepts of a NoDB paradigm [2], we developed EpivizFileServer [10], a Python library that implements an in-situ data query system for local or remotely hosted indexed genomic files, not only for visualization but also data manipulation. Our design of the File Server library was based on the following goals:

- Efficiently parse minimal necessary bytes from an indexed genomic file to query data for a specific genomic region
- Define transformations and summarizations directly over files and lazily compute these at query time
- Scale operations to concurrently process multiple file query and transformation requests
- Implement cache over files for faster access and improve repeat query performance
- REST API for developers and bioinformaticians to build interactive visualization and exploratory tools over genomic data stored in flat files
- Integration with existing bioinformatic tools and software to interactively visualize and explore genomic data directly from files

The File Server [10] library decouples data from analysis workflows and provides an abstract interface to define computations independent of the location, format or structure of the file. Our major contribution on this research project was to efficiently and intuitively define transformations and summarizations directly over files, without the hassle of downloading the files locally or preprocess for exploratory data analysis. Using the library, researchers and analysts can author shareable and reproducible data exploration workflows in an intuitive and programmatic way. If the files are hosted on a public server, the library requires the server hosting the data files to support HTTP range requests [1], so that the parser can only request the minimum necessary byte-ranges needed to process the query. The library caches the index and previously queried data from the file to reduce the numebr of requests, and cleans the cache when they have not been accessed in a period of time. The library supports various file formats - BigBed, BigWig and any tabular file that can be indexed using Tabix. Once these data files are described, users can define summarizations and transformations on these files using NumPy (or NumPy-like) functions.

The File Server [10] library uses Dask [21] to scale and concurrently process multiple queries and transformations over files. The cache implementation makes sure we only access bytes not already accessed and stored locally. Developers of bioinformatic tools and systems can use the Library's REST API to build interactive data visualization or exploratory tools over files.

The system works well for moderate sized repositories to apply complex transformations over files at query time. This depends on available system resources (processors, memory) for Dask to scale and the location of files (local or remote). Our tests on transformations show that latency of the system increases as we apply transformations over an increasing number of files. The parser module spends the majority of time reading the index tree to find positions within the file that contain the data. For example applying a transformation over 20 files, the system has to individually parse the index of the 20 files to get the data and then apply transformations. The generic way to accomplish this task is to search through the index of each file and locate the target interval. To make this process faster, as latency is important in interactive visualization tools, we need a method that scales interval search well for large number of files.

2.2 Quadtree

2.2.1 Construction. To construct an index that stores information of multiple files at the same time, we project it into a structure that allows interval overlaps while still giving a fast search time. A traditional R-tree would not suffice as it relies on linear continuity when binning, and insertions from different files would trigger the balancing. Rebalancing the R-tree might be computationally expensive. This restricts the use of this data structure as an index in dynamic systems where we build an index on the fly as queries are processed.

To avoid the issue of rebalancing, we explored other space partitioning data structures for indexing genomic intervals. Quadtree is a hierarchical spatial data structure that does not need rebalancing, and supports both point and interval data. Quadtree subdivides space into hypercubes for indexing and quick retrieval of data. There are other Balanced Box-Decomposition Trees [19] that also achieve similar build time and space usage as a Quadtree. For this paper we will focus on using Quadtree.

Since our input to build this index is genomic intervals stored in files, we need to map intervals into a 2D space and index this space with Quadtree. To map linear intervals to a 2D space, a popular known technique in mathematics is Space filling curves. Hilbert Curve is a space filling curve that maps linear data to multi-dimensional space [8]. In this paper we will only be exploiting 2D Hilbert Curve to map interval data, but it is possible to apply the mapping to more complex data. The benefits of using Hilbert curve than other space filling curves such as Gosper curve are that 1)

it preserves proximity as data close linearly are also close after mapping and 2) four hilbert curves of level i constructs a hilbert curve of level $i + 1$ which can be utilized by Quadtree to query efficiently.

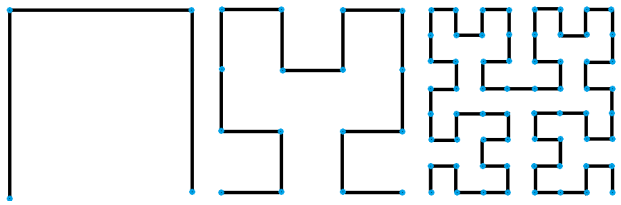

Figure 2: Hilbert curve of level 1, level 2 and level 3 (from left to right). Note that Hilbert curve of level 0 is composed of only 1 point.

Definition 2.1. Each reference **genome** G of an organism contains several **chromosomes** C of varying length. When sequencing experiments are run on a sample DNA, the reads are mapped to the reference genome for various types of functional and sequence analysis. This results in several **genome files** F_i. Each chromosome C has a maximum **length** L for a given genome G.

Definition 2.2. A File F contains **non overlapping intervals** I_i mapped to chromosomes using the reference genome.

Definition 2.3. A **data interval** is an interval $I = [start, end]$, $start > 0, end > start$ associated with a file id and byte location. The file id and byte location is metadata that the File Server [10] uses to get data.

Definition 2.4. A query that is sent into the Quadtree is a **search interval** that is defined as $I = [start, end], start > 0, end > start$.

Definition 2.5. A **hypercube** or a **bounding box** is a list of coordinates in R^d that defines an enclosed space.

Definition 2.6. Two hypercubes or bounding boxes, namely A and B, **intersect** if and only if $A \cap B \neq \emptyset$.

Definition 2.7. A Hilbert curve of level i contains exactly 4^i points.

Definition 2.8. A Quadtree node n at level i is **appropriate** for a hypercube of a data D only when the hypercube of n, namely C, fully contains D and any hypercube of the children of n does not fully contain D. In other words, $C \supset D$ and $\forall c \in$ {hypercubes of the children of n}, $c \not\supset D$.

Definition 2.9. A **minimum bounding rectangle** of rectangles in S is the smallest rectangle R that contains all the rectangles. In other words, $R =_{min}$ rectangle $\supseteq \cup_i r_i, r_i \in S$

Proposition 2.10. *For a hilbert curve of level $i \geq 0$, the number of points between its starting point and the other end of the diagonal containing the starting point is* $(\sum_{k=0}^{i-1} 2 * 4^k) + 1$.

The proof of this can be done in very simple induction on the levels, which will be omitted here.

Proposition 2.11. *Given a chromosome length L, the minimum Hilbert curve that contains L is of level* $\lceil log_4(L) \rceil$.

Proposition 2.11 follows immediately after Definition 2.7, as the number of points in a level i Hilbert curve is 4^i. When building the index, we first compute the maximum level of the Hilbert curve that contains the entire chromosome C, and we use it as the Quadtree indexing space. Since the Hilbert space's side is always a multiple of four, it naturally splits the space into the nodes of quadtree.

We convert the start and end index of the intervals into Hilbert curve coordinates, and store the bounding boxes based on only these two coordinates of each interval. When a node reaches the maximum entries it can hold, it splits the space into 4 quadrants and reshuffles the data into the appropriate children nodes, which is defined earlier. The depth that the data belongs to should be deep enough to ensure less total searched data but also maintain enough precision. Once the index is constructed, it can be stored to disk for portability and compressness. Our methods also support on-disk based index query operations since generating this index for a large number of files may not always be fast.

2.2.2 Query. The search interval is first converted into an augmented hypercube, then passed into the Quadtree. Since we only use the start and end index of a data interval when constructing, it does not truly represent the shape of the data. Adding the augmented shape calculation into the construction phase creates too much overhead for each data point. Instead we calculate the augmented hypercube of the search interval by padding different levels of Hilbert curves into the minimum bounding rectangle. By Proposition 1, we can seperate the search interval into different levels of hilbert curves, get the diagonal of each hilbert curve and use the minimums and maximums of x, y coordinates of the diagonals to acquire the padded bounding box. The padding calculation can be considered as a constant time when the level of Hilbert curve that is used to build the Quadtree is fixed.

When a search interval is inputted into the Quadtree, the Quadtree first searches the current node, then searches all the children of this node that it intersects (as in definition 2.6) iteratively.

Proposition 2.12. *The search method guarantees to find all data intervals that intersects the search interval.*

There are 3 types of relationships between the search interval and the data interval:

- **The search interval covers the whole data interval.** In such a case, the data rectangle is guaranteed to be inside the searching bounding box as the searching box is padded to cover every data point in its interval.
- **The search interval intersects the data interval at one end.** Thus the endpoint that intersects must be in the search interval, then it is also guaranteed to be found.
- **The search interval is smaller than the data interval.** We claim this still guarantees finding the data. As the search is traversing down the tree, it searches all the nodes that overlap with the searching interval. Since the data interval is larger, although it might have a weird shape, it will still be at a precision level where it overlaps the search interval.

Thus, this searching guarantees to find all data intervals that overlap the search interval.

Proposition 2.13. *The Quadtree takes time O(nlogn) to construct, takes O(logn) time to perform a search and uses O(n) space, taking the max Hilbert curve level of the quadtree and the dimension as constant.*

Since our implementation of the Quadtree is almost similar to the standard Quadtree data structure, We expect the construction time and query time to be very similar to the standard Quadtree [19].

3 RESULTS

All tests were run on a standard Amazon AWS EC2 (t2.xlarge) instance with 4 vCPUs and 16GB memory. To evaluate the impact of cache on the system, we randomly generated 20 different genomic range queries and repeatedly queried these against the Web server for 60 seconds. We use wrk (https://github.com/wg/wrk), a HTTP benchmarking tool capable of generating significant load to test the API. We run the tool on its default settings using 2 threads and 10 connections concurrently to send requests to the system.

We run the system on two different modes, one with the cache feature and the other without the cache. In the cache implementation, if the given genomic region already exists in the cache, it is fetched quickly and sent back to the user whereas in the non-cache setting, the library always parses the file to query data for the given genomic range. Since local file access is fast, our results are comparable between the cache and non-cache settings. Instead, we hosted the files on a S3 object store bucket at University of Maryland, adding network latency to the system. The results indicate the cache implementation significantly improves the performance of the system. Table 1 displays the results of these tests. To make sure other processes are not interfering in the benchmarking process, we disabled the serialization process for file objects discussed in the methods section.

We also ran a similar experiment to compare the performance of EFS with existing bioinformatic tools. We use the PyBigWig python package [18], an extension to the C library, libBigWig that can read or parse local or remote BigWig and BigBed files. We run the same experiment as before, where we generate 20 random genomic ranges and execute these queries repeatedly for 60 seconds using the PyBigWig library. We read the same remotely hosted file and measure the average time per query and calculate standard deviation. We notice that EFS performs significantly faster if the file is hosted remotely. Unsurprisingly direct access to the local file using PyBigWig is significantly faster compared to EFS. The extra overhead is EFS is due to the 1) use of an intermediate representation such as Pandas DataFrame [16] so that transformations or summarizations can be performed across/within files and, 2) using a portable data transfer representation of the results (JSON) so that multiple clients can query the system. PyBigWig only reports the intervals and the data in those intervals.

To compare our indexing method, we downloaded the entire epigenomic roadmap dataset, a total of 1032 files. We extract the intervals from these files and build a Quadtree index over these intervals. We randomly choose 100 files to query for intervals in a given region over the Quadtree index. We compared our indexing method against the querying from 100 files using the File Server [10]. We ran the same queries on 100 files using a File Server [10] with disk cache and memory, and also a file-based Quadtree index. The result indicates that the Quadtree improves query time significantly, which is presented in Figure 3. Not only does Quadtree improves query time, it also decreases the size of the indices, which is presented in Table 2.

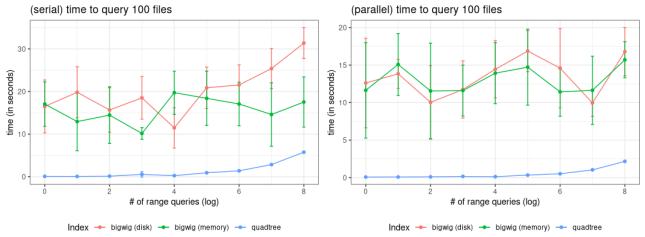

Figure 3: These plots displays how EFS and Quadtree perform on queries on large numbers of files. Although there are a very large variance on the performance of EFS, Quadtree handles the queries much faster than the naive approach.

4 FUTURE WORK

Single cell technologies generate large datasets measuring tens of thousands of features over thousands of single cells. Although very efficient to query by genomic region, if we are only interested in a few cells from such large matrices, the EFS library using the tabix format still has to parse the entire row and filter the columns. Tabix is a commonly used indexing technique for any tabular genomic data set (the first three columns must be chromosome location, start and end). Interactive analysis, including visualization of these datasets is a challenging task especially for queries to filter by columns (or cells) and efficiently transferring these long matrices between server and client. Piccolo et al. 2019 recommend using coordinate based fixed width formats as a fast and scalable approach to query tabular genomic data. In addition, the genomics community has been using HDF5 based formats AnnData or H5ad [23] and loom (https://loompy.org) to store large genomics datasets and metagenomic datasets [22]. We are currently exploring ways to efficiently query HDF5, H5ad or loom format files. These formats do not natively support remote querying like BigWig or Tabix but require an additional server like h5serv (https://support.hdfgroup.org/pubs/papers/RESTful_HDF5.pdf) setup for Web queries.

Recent approaches like Pyranges [20] have implemented data structures for efficiently manipulating genomic intervals in Python. The EFS library can be extended to incorporate these file formats and data structures to efficiently query, perform interval overlap operations and interactively explore multi-omic datasets.

We would like to compare our Quadtree indexing approach with existing interval indexing structures like Nested Containment Lists [3], and also the naive File Server [10] naive approach in a more thorough and analytical manner. Adopting the compressed Quadtree [7] construction would reduce the size by a little. We would like to extend the File Server library to use the Quadtree as an index over files when it queries for data.

Table 1: Impact of Cache on Processing Requests

Implementation	Avg Latency (in ms) (± SD)	Requests (per sec) (± SD)
EFS – No Cache (remote file)	1152 (± 201.32)	8.2 (± 0.44)
EFS – Cache (remote file)	68.41 (± 83.55)	179.4 (± 3.2)
EFS (local file)	36.05 (± 8.74)	284.1 (± 41.31)
PyBigWig (remote file)	121.864 (± 40.67)	–
PyBigWig (local file)	0.52 (± 0.28)	-

Table 2: Comparison of File Size

Implementation	Indexes on Chromosome 11 of a file
Precomputed Quadtree Index	318.3 MB
BigWig R-tree Index (Lower Bound Estimate)	~ 4 G

4.1 Conclusion

Based on the concepts of a NoDB paradigm, we present a file-based Python library, an in-situ data query system for indexed genomic files, not only for visualization but also transformation. The library implements several features provided by a traditional database system for query, transformation and caching. We also presented a new approach to index and query large amounts of genomic data, which improves our EFS performance. We discussed new research approaches to build a comprehensive file-based data visualization and exploration system for genomics datasets.

ACKNOWLEDGMENTS

This thesis and work is finished with the help and supervision of Dr. Hector Corrada Bravo and Jayaram Kancherla. Without them, the goal of the work or paper would not have been realized.

I want to express my appreciation to my supervisor, Dr. Hector Bravo. It has been my honor to have him as my first advisor in academia. My experience and career in research would not have started so smoothly without him. Thank you so much for the guidence over the past 3 years.

I would also like to express the same amount of gratitude to my advisor and friend, Jayaram Kancherla. He has provided help in and beyond research, and also enlightens in life.

I want to thank Dr. Mohamed Gunady, Domenick Braccia and other members of CBCB. The fun small talks and meetings really substantiated me through my college. I wish these memories would last forever.

Lastly, many thanks for my family who supports me on my pursue of such a carrer.

REFERENCES

[1] Accessed 2019. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. https://tools.ietf.org/html/rfc7233

[2] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2015. NoDB: efficient query execution on raw data files. *Commun. ACM* 58 (11 2015), 112–121. https://doi.org/10.1145/2830508

[3] A. V. Alekseyenko and C. J. Lee. 2007. Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics* 23, 11 (Jun 2007), 1386–1393.

[4] Florin Chelaru, Llewellyn Smith, Naomi Goldstein, and Héctor Corrada Bravo. 2014. Epiviz: interactive visual analytics for functional genomics data. *Nature Methods* 11, 9 (01 Sep 2014), 938–940. https://doi.org/10.1038/nmeth.3038

[5] Carrie A. Davis, Benjamin C. Hitz, Cricket A. Sloan, Esther T. Chan, Jean M. Davidson, Idan Gabdank, Jason A. Hilton, and et al. 2017. The Encyclopedia of DNA elements (ENCODE): data portal update. *Nucleic Acids Research* 46, D1 (June 2017), 36–44. https://doi.org/10.1093/nar/gkx1081

[6] The Cancer Genome Atlas Research Network et al. 2013. The Cancer Genome Atlas Pan-Cancer analysis project. *Nature Genetics* 45, 10 (01 Oct 2013), 1113–1120. https://doi.org/10.1038/ng.2764

[7] T. Gagie, J. I. González-Nova, S. Ladra, G. Navarro, and D. Seco. 2015. Faster Compressed Quadtrees. In *2015 Data Compression Conference*. 93–102.

[8] David Hilbert. 1935. *Über die stetige Abbildung einer Linie auf ein Flächenstück*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–2. https://doi.org/10.1007/978-3-662-38452-7_1

[9] Wolfgang Huber, Vincent J. Carey, Robert Gentleman, Simon Anders, Marc Carlson, Benilton S. Carvalho, and Hector Corrada Bravo , et al. 2015. Orchestrating high-throughput genomic analysis with Bioconductor. *Nat. Methods* 12, 2 (Feb 2015), 115–121.

[10] Jayaram Kancherla, Yifan Yang, Hyeyun Chae, and Hector Corrada Bravo. 2019. Epiviz File Server: Query, Transform and Interactively Explore Data from Indexed Genomic Files. *bioRxiv* (2019). https://doi.org/10.1101/865295 arXiv:https://www.biorxiv.org/content/early/2019/12/05/865295.full.pdf

[11] Jayaram Kancherla, Alexander Zhang, Brian Gottfried, and Hector Corrada Bravo. 2018. Epiviz Web Components: reusable and extensible component library to visualize functional genomic datasets. *F1000Research* 7 (17 Jul 2018), 1096–1096. https://pubmed.ncbi.nlm.nih.gov/30135734 30135734[pmid].

[12] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. 2002. The human genome browser at UCSC. *Genome Res.* 12, 6 (Jun 2002), 996–1006.

[13] R. M. Layer, B. S. Pedersen, T. DiSera, G. T. Marth, J. Gertz, and A. R. Quinlan. 2018. GIGGLE: a search engine for large-scale integrated genome analysis. *Nat. Methods* 15, 2 (02 2018), 123–126.

[14] Heng Li. 2011. Tabix: Fast Retrieval of Sequence Features from Generic TAB-Delimited Files. *Bioinformatics (Oxford, England)* 27 (03 2011), 718–9. https://doi.org/10.1093/bioinformatics/btq671

[15] BP. Maintainer. 2019. ExperimentHub: Client to Access ExperimentHub Resources. https://doi.org/10.18129/B9.bioc.ExperimentHub.

[16] Wes Mckinney. 2010. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference* (01 2010).

[17] Martin Morgan, Marc Carlson, Dan Tenenbaum, Sonali Arora, and Lori Shepherd. 2019. AnnotationHub: Client to access AnnotationHub resources. https://doi.org/10.18129/B9.bioc.AnnotationHub.

[18] Devon Ryan, Gökçen Eraslan, Björn Grüning, Ricardo Silva, Patrick Marks, Fidel Ramirez, and Edward Bett. 2019. deeptools/pyBigWig: 0.3.17 (Version 0.3.17). https://dask.org

[19] Hanan Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., USA.

[20] E. B. Stovner and P. Sæ trom. 2020. PyRanges: efficient comparison of genomic intervals in Python. *Bioinformatics* 36, 3 (Feb 2020), 918–919.

[21] Dask Development Team. 2016. Dask: Library for Dynamic Task Scheduling. https://dask.org

[22] Justin Wagner, Florin Chelaru, Jayaram Kancherla, Joseph N Paulson, Alexander Zhang, Victor Felix, Anup Mahurkar, Niklas Elmqvist, and Héctor Corrada Bravo. 2018. Metaviz: interactive statistical and visual analysis of

metagenomic data. *Nucleic Acids Research* 46, 6 (02 2018), 2777–2787. https://doi.org/10.1093/nar/gky136 arXiv:https://academic.oup.com/nar/article-pdf/46/6/2777/24526301/gky136.pdf

[23] F. Alexander Wolf, Philipp Angerer, and Fabian J. Theis. 2018. SCANPY: large-scale single-cell gene expression data analysis. *Genome Biology* 19, 1 (06 Feb 2018), 15. https://doi.org/10.1186/s13059-017-1382-0