Journal Title Here, 2024, pp. 1–6

doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

PAPER

ZygosDB: An efficient read-only database for Genome-Wide Association Studies (GWAS)

Nick van Luijk[1,*]
Supervisors: Marcel Reinders[1], Niccolo Tesi[1]

[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

This paper describes ZygosDB, a novel and efficient read-only database designed specifically for querying positional genomic data required for Genome-Wide Association Studies (GWAS). ZygosDB addresses limitations of existing solutions like Tabix by offering optimized data structures, compression techniques, and parallel query execution.

Our evaluation shows that ZygosDB achieves a significant speedup of 2-5 times in query throughput compared to Tabix. This improvement comes from our focus on efficient data storage and retrieval tailored to the specific needs of GWAS.

The paper also explores the impact of multi-threading on query performance and the role of compression algorithms in optimizing query throughput. We identify a decrease in performance beyond a certain number of threads and discuss the influence of compression algorithms like Gzip and LZ4.

While ZygosDB offers substantial performance gains, future work should explore avenues for further optimization, such as measuring query latency, refining memory usage, and investigating specialized column support. Overall, ZygosDB establishes itself as a powerful tool for efficient querying of large genomic datasets, facilitating more effective GWAS.

Key words: database, bioinformatics, genome, gwas

Introduction

Unless you have an identical twin, your DNA differs from everyone else's DNA. This molecule, which you can find in the nucleus of all cells of an organism, contains all genetic information required to develop an organism, keep it alive and allow it to reproduce. DNA can be represented as a long string of nucleotides, the set of A, C, T and G bases. When comparing your DNA with a reference genome, you will discover genetic variations between you and the reference. When a single base in a row has been substituted with a different base, we call this a Single Nucleotide Polymorphism (SNP). Structural Variants (SVs) are larger variations in the genomic structure, where multiple consecutive bases have been deleted, duplicated or rearranged. Genome-wide association studies (GWAS) are studies that try to find associations between SNPs, SVs and genetic traits. To help researchers find more of such associations, visualisation tools such as snpXplorer have been created [1] Accessing and querying the vast amounts of data required by these visualisation tools, however, takes a significant amount of time.

The data to be queried consists of millions of rows. Each row belongs to a specific chromosome and a positive integer indicating the position, relative to the start of the chromosome. For some datasets, it is possible for multiple rows to exist at the same position in the same chromosome. The position can therefore not be used to uniquely identify rows. Along with the position, rows can store an arbitrary amount of columns storing integers, floating point numbers and strings. Some strings, such as "NA", can occur very frequently within the datasets. Common queries consist of a chromosome, a start position and an end position. These queries then return all rows with a position that lies in this interval. Rows could consist of the names of SNPs as strings and the probabilities (P-values) of the corresponding SNPs occuring as floating point numbers.

The current implementation of snpXplorer uses Tabix to query the datasets. Tabix is a "generic tool that indexes position sorted files in TAB-delimited formats such as GFF, BED, PSL, SAM and SQL export, and quickly retrieves features overlapping specified regions" [2] It generates indexing files that can later be used to quickly query data from tab-separated value files.

Despite its relatively fast and flexible nature, Tabix supports querying only one file at the time. After parsing Tabix's query results, snpXplorer joins the query results. Additionally, Tabix requires the full tab-separated value file, which can be gigabytes in size, to be present on the computer, even if not all columns are queried. Furthermore, as Tabix provides no official Python module, and the third-party Python module pytabix[1] has not received updates since 2015, snpXplorer currently uses the Tabix command-line interface as a child process and parses the output, the overhead of which could be significant.

As an alternative to tab-delimited text-based formats, binary formats specialized for storing biological data have been created, such as BigWig and BigBed [3] These formats can however not be used for storing the datasets snpXplorer requires, as the formats only support a predefined set of columns with dedicated roles, rather than an arbitrary amount of columns with arbitrary types. For example, P-values cannot be stored in BigWig and BigBed, as they do include a column dedicated to storing P-values. [3, Supplementary Table 1]

To implement a custom read-only database, we looked at existing implementations of SQLite [4] and Tabix. SQLite is a general-purpose SQL database engine that uses a single process to interact with a database file saved locally the device [5] Tabix is a tool that creates index files to quickly query TAB-delimited files that contain data with chromosomal positions [2] Examining how SQLite and Tabix operate under the hood provided an understanding of the mechanics of efficient data storage and retrieval.

We created ZygosDB to more efficiently query data required for finding and visualising interactions between SNPs and SVs.

The main question to be answered is as follows:

How can an efficient read-only database be designed and implemented for querying chromosomal, positional data?

This comes with a list of related sub-questions that aim to help answer the main question:

1. What binary database format exist and what can we learn from them?
2. How can positional, genomic data be stored and queried in a custom binary database format?
3. How can query throughput be increased by compressing the data?
4. In what ways can querying data be parallelized?

[1] `https://github.com/slowkow/pytabix`

The main contribution of this research is a new and efficient read-only database for querying all data required by snpXplorer.

Methodology

Implementation

The Rust programming language was chosen as the language to implement the database in, because of its efficiency, memory-safety and lack of garbage collector[6] To allow bioinformaticians to more easily use our database, a python wrapper was created using PyO3[2]. Rust supports most popular platforms, including x86_64 Windows, x86_64 Linux and ARM64 Linux and x86_64 macOS. Additionally, Rust programs can also be compiled to WebAssembly [7] allowing the database to be queried directly in web browsers.

Optimisations

After implementing my read-only database, the next steps were to optimise it. Two main ways of optimising for query throughput were looked at; compression and parallelisation.

By compressing the database when it is written to disk, the aim is to reduce the amount of bytes the program has to read while querying. The hypothesis is that the overhead of decompressing the data is smaller than the time gained by reading less bytes while querying. An additional benefit of compressing the database is that the database takes up less storage space. The input datasets can be tens of gigabytes in size, so any reduction of required storage size is nice. Support for using compression algorithms Gzip and LZ4 has been implemented.

Another way to optimise the querying is by reading the database in parallel. As SNPs and SVs are stored in different tables, queries that require searching through multiple tables can be done in parallel. Additionally, decompression and deserialisation of entries can be divided over multiple threads for queries that return a range of entries. Both of these parallelisations should increase the query throughput, but to discover if this is actually true and by how much, the query throughput should be measured.

Bridging Tabix and Python

To accurately measure real-world performance, all benchmark tests require the query results to be readable using Python. As there is no up-to-date Python package to interface with Tabix, we resorted to running Tabix as a child process. This is how the snpXplorer currently integrates Tabix too. The output of the Tabix process is then converted to a Python string object and passed to the `pandas.read_csv` function of the Pandas library[3]. The resulting `DataFrame` then contains the queried data to be used in other parts of a program.

[2] `https://pyo3.rs/`

[3] `https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html`

Benchmarking

To measure how the optimisations affect the query throughput, test have been run on a dataset of approximately 570 megabytes provided by our supervisor, containing SNPs of over a million people with Alzheimer's disease[8] The database and the Tabix index files were built prior to running the tests.

For the first test, the goal was to measure how many rows can be read per second for different interval windows. We've measured the sustained query throughputs for a duration of 60 seconds with window sizes increasing exponentially. The sizes of the windows were 1, 10, 100, 1000, 10 000, 100 000 and 1 000 000. In an attempt to eliminate CPU instruction cache misses and branch prediction rollbacks, all tests had a 60 second "warmup" period, where queries were being executed, without their results counting towards the benchmark. Only after this warmup, the actual query throughput was being measured. These measurements were done for Tabix and our database, with multiple configurations of our database; no compression, Gzip compression, Zlib compression, LZ4 compression and Zstandard compression. For all configurations, Tabix and our database used a single thread.

For the second test, the goal was to research how increasing the number of threads influences the query throughput. The sustained query throughputs were measured for a duration of 60 seconds, again with a warmup of 60 seconds. The tests were run with thread counts of 1 through 32, increasing by steps of 1. Just as in the first test, these measurements were done for Tabix and our database, with multiple configurations of our database; no compression, Gzip compression, Zlib compression, LZ4 compression and Zstandard compression.

Execution details

The tests were run on a desktop computer using an AMD Ryzen™ 9 7950X3D[4] with 16 cores and 32 threads, 64 gigabytes of RAM (G.SKILL Trident Z5 Neo RGB DDR5-6000 CL30-38-38-96[5]) and with the datasets, databases and indices stored on a Samsung 990 PRO PCIe 4.0 NVMe M.2 SSD[6]. As Tabix does not support running on Windows, all tests were run on Ubuntu 22.04.3 LTS using Windows Subsystem for Linux (WSL 2). To reduce the overhead of WSL, all datasets, indices and databases were moved from `/mnt/c/` to the home directory of the current Linux user.

Database implementation

Building from a configuration file

Our database supports a variety of configurable options, which must all be specified in a configuration file. The path to the datasets, which chromosomes are in the dataset and the columns with their corresponding types are required to be declared in this configuration file.

A configuration file was chosen over specifying command line arguments, because the large amount of command line arguments that would have to be specified, would make reading and debugging the command difficult. Additionally, it is easy to forget which command has previously been used to build a database in case it must be rebuilt in the future.

Tom's Obvious Minimal Language (TOML)[7] was chosen to create the configuration file in, as it is commonly used in both the Rust and Python ecosystems. Additionally, the `toml` crate natively supports showing exactly where mistakes in the configuration file are made, which improves the usability of our database.

Database header

At the start of the binary file, our database, the header is placed. This header starts with a magic value of `5A 79 67 6F 73 44 42` in hexadecimal, or "ZygosDB" when interpreted as text. This magic value is always verified when a database is opened, to ensure the developer using our database did not mistakenly open the wrong file. Additionally, when opening the built database using a hex editor, it is revealed that this is a ZygosDB database, and not arbitrary bindary data.

After the magic value, the version number is found. This version number is used to prevent outdated versions of the Python wrapper trying to read future versions of the binary database format. A version mismatch could result in reading incorrect data or crashes.

The database header also stores general information about the datasets, such as the names and types of columns, and the locations of the table indices.

Sequential access

Reading sequentially is generally much faster than reading the same data in a random order. By accessing data in a random order, seeking to the correct position is required. This takes more time than reading data following the current position. While this is less of a problem for data stored on solid state drives (SSDs), randomly reading data stored on Hard Disk Drives (HDDs) is an order of magnitude slower due to having to physically move the head in the drive.

Additionally, when data is read from disk, only whole pages can be read. This means that to read a single row, an entire page of typically 4096 bytes is read from disk and stored in memory. While deserializing the bytes stored in memory, even smaller sections of this data are copied to the internal cache of the CPU. Accessing memory cached in the CPU is another order of magnitude faster than when bytes have to be read from memory. By deserializing a large chunk of sequentially stored rows, the chance that this data is already stored in memory or even the CPU cache is significantly higher than deserializing the same rows in a randomized order.

The decision was therefore made to store all data in a sequential order for our database. When an input dataset is read, only the cells of columns specified in the configuration file are deserialized. These cells are then grouped into rows, after which the rows are sorted by their position. The sorted dataset is then serialized and written to disk.

To query a range of rows from the database, only the location of just the first row of the result must be known. All following rows are namely found directly after this first row. This means only one seek operation is required to deserialize all resulting rows.

[4] `https://www.amd.com/en/products/processors/desktops/ryzen/7000-series/amd-ryzen-9-7950x3d.html`

[5] `https://www.gskill.com/product/165/390/1661410135/F5-6000J3038F16GX2-TZ5NR`

[6] `https://www.samsung.com/uk/memory-storage/nvme-ssd/990-pro-2tb-nvme-pcie-gen-4-mz-v9p2t0bw/`

[7] `https://toml.io/`

Variable length integers

All rows store an integer position relative to the start of the chromosome. As chromosome 1, the longest human chromosome, contains around 250 million nucleotides, all possible values of the position column fit inside of 32 bits. A significant portion, all rows with a position of less than $2^{24} - 1 = 16777215$ fit inside a 24-bit unsigned integer, saving one byte per row.

To reduce the amount of bytes required per row, and potentially increasing query throughput, integers with a variable length are used to store the position. This is implemented using the encoding found in the vint64 crate[8]. This losslessly stores any signed or unsigned 64-bit integer in at most 9 bytes, where the smallest amount of required bytes is always used.

If the query throughput is actually increased is tested later in Results.

Indexing using B-trees

SQLite uses B-trees to store pages of data and indices in a database file[9] This data structure is very similar to a binary search tree, but B-trees can contain more than two children per node [10] Both binary search tries and B-trees have a time complexity of $O(logn)$ for lookups, where n is the amount of nodes in the tree. In practice, however, B-trees are faster, as they do more comparisons per node, resulting in less traversed nodes than a binary search tree would have required.

In our database, B-trees are used in querying the index of a table. When building the database, all rows of data are first serialized and written to the database file. Every user-configurable n rows, a new index is created and appended to a list, creating ”blocks” of size n. This index stores the position of the next row, relative to the start of the file, together with the chromosomal position. After all rows have been written, the list of indices is serialized and appended to the database file.

To query the database, the list of indices is read from the database file, deserialized and stored in-memory as a B-tree. Loading the B-tree into memory has to be performed only once, after which any number of queries can be performed. This is implemented using Rust's `std::collections::BTreeMap`[9].

In order to perform a ranged query that returns all rows with a position between a start and end, the B-tree of the index is used to find the block that contains the first row within this range. To find the first row, we first deserialize the cell containing the position. If this position is smaller than the position of the row we're looking for, we skip deserializing the remaining cells of the current row. This process of deserializing the position and skipping the remaining cells is repeated until the correct row is found. Once this row is found, all sequential rows are fully deserialized and appended to a list. The Python wrapper then takes this list and converts it to a list of Python objects, ready to be used by the developer using our database.

Compressing the data

Some datasets used by bioinformaticians can be hundreds of gigabytes in size when not compressed. To make these datasets easier to handle, it is essential that the data can be compressed before it is stored the database.

Database compression is implemented in our database by compressing all serialized rows between two indices (a block). While querying, any block that must be read is first decompressed, after which the decompressed rows are deserialized like previously described. Because the data must be decompressed, additional overhead is introduced. The effects of this are measured in Results

Querying in parallel

The database can be queried in parallel in two separate ways.

In the first option, multiple queries can be performed at the same time. As our database cannot be written to, it is impossible to get race conditions. The developer using our database can therefore open the database multiple times in separate threads, and perform multiple queries at the same time.

In the second option, the work of performing a single query is split up over multiple threads running in parallel. At the start of a query, before rows can be deserialized, the position of the rows must first be looked up in the index. For a range query with a large distance between the start and the end, multiple blocks have to be decompressed and deserialized. Because the locations where these blocks start are all found in the index, this work can be performed on multiple threads at the same time. This is implemented in our database and exposed through the Python wrapper. How this affects the query throughput is found in the next chapter.

Results

Overview of ZygosDB

We developed ZygosDB, an efficient read-only database to query positional, genomic data. As discussed in section 3, ZygosDB is built from a configuration file into a binary file. All input datasets consist of multiple files, one for each chromosome in the dataset. To store these files in the database, the contents are parsed and converted to variable-length integers, floating point numbers and strings of text, before they are serialized. This serialized data is also known as a table. The tables are then optionally compressed using Gzip, Zlib deflate, LZ4 or Zstandard, before they are written to disk. An index is appended directly after each serialized table.

To query the database, the indices are read and converted to B-trees. The position in the database of the start of the serialized data can then efficiently be looked up using the B-trees. Once this position is found, data is read from disk and copied into memory, before it is decompressed and deserialized.

Parallelisation is achieved by distributing the work of a single query over multiple threads. The index of a table stores the positions of ”blocks”. Each block consists of a user-configurable amount of rows, specified in the configuration file used to build the database. Using the B-tree, it is possible to look up which blocks of rows contain the query results. The work of retrieving the blocks, decompressing and deserialising them, is distributed over multiple threads to reduce the amount of time a single query takes, increasing the throughput.

Benchmarking analysis

The benchmarking analysis focuses on evaluating the performance of ZygosDB and comparing it against Tabix, the

[8] `https://crates.io/crates/vint64`

[9] `https://doc.rust-lang.org/std/collections/struct.BTreeMap.html`

tool currently in use by snpXplorer. The analysis includes two main tests: measuring query throughput for different window sizes and examining the impact of multi-threaded optimization on performance.

Query throughput

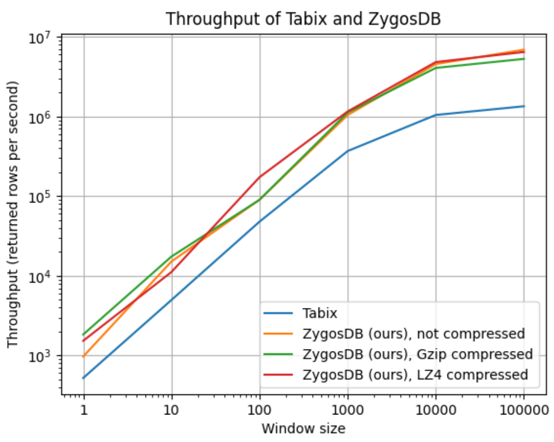

Fig. 1. The query throughput, measured in the amount of rows returned per window size, for both Tabix and our database.

As visible in Figure 1, ZygosDB is faster than Tabix for all window sizes. As the window size increases, the query throughput increases as well.

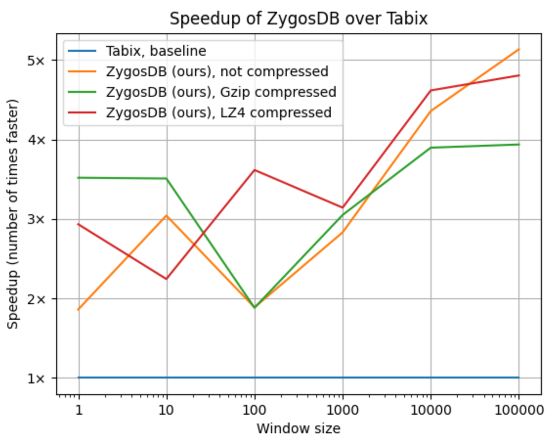

Fig. 2. The speedup of ZygosDB over Tabix.

From Figure 2 we can see ZygosDB is approximately 2-5 times faster than Tabix. Notable is the positive correlation between the window size and the speedup.

An exception to this is visible in the tests without compression and with Gzip compression. Here, in the jump from a window size of 10 to a window size of 100, the speedup actually decreases. The speedup does increase again when going from a window size of 100 to a window size of 1000, but the speedup at a window size of 1000 remains below the speedup at a window size of 10. In the tests with LZ4 compression, the speedup actually increases compared when going from a window size of 10 to a window size of 100, after which it drops when reaching a window size of 10000.

Multi-threaded optimisation

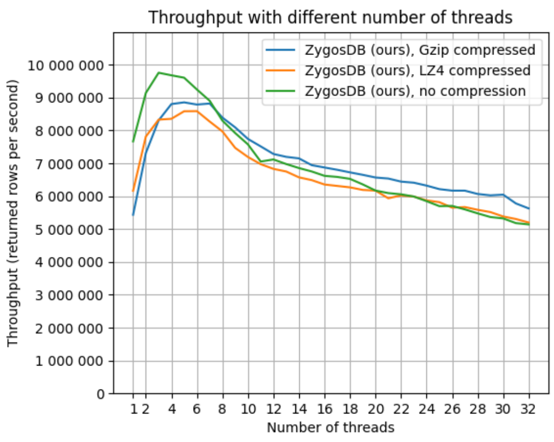

Fig. 3. The multi-threaded query throughput with a window size of 100 000, measured with thread counts of 1-32.

To measure how splitting up queries and resolving them on multiple threads affects the throughput, tests with a varying amount of available threads were ran. Figure 3, Shows that initially, as the number of threads increases, the throughput also increases.

However, at a thread count of around 4 threads, the throughput no longer increases. With higher thread counts, the throughput starts decreasing. With 32 threads, the throughput is actually lower than when using only 1 thread.

Additionally, with thread counts of 7 or less, not using any compression is faster than using Gzip or LZ4 compression. With thread counts of 8 and above, Gzip compression is faster than not using compression or using LZ4 compression.

Responsible Research

The repository containing the source code of our database, the Python wrapper and the scripts used for running the benchmarks, is publicly available on GitHub at `https://github.com/TechnologicNick/zygos_db` This allows anyone to reproduce the benchmark results, generate the graphs and verify our findings.

Discussion

As ZygosDB is more specialised than Tabix, we expected the query throughput to be higher. We are happy with the achieved speedup of approximately 2-5 times.

Why the speedup of ZygosDB with Gzip compression and without compression goes down around a window size of 100, visible in Figure 2, is unknown to us. Queries with a window size of 100 have a higher probability of being spread over multiple blocks than queries with a window size of 10, because blocks of 1000 rows were used in these tests. However, this does not explain why ZygosDB with LZ4 compression with a window size of 100 is significantly faster than using a window size of 10, as this theory would suggest the opposite.

The initial increase in query throughput when increasing the amount of threads is what we expected. We expected that the throughput would at some point me met with diminishing returns, stagnating at a high throughput. Instead, the throughput started dropping. This is most likely due to the additional overhead of distributing the work across multiple threads, waiting for them to all be done and then aggregating the results.

Conclusions and Future Work

In this research, we aimed to design and implement an efficient read-only database tailored for Genome-Wide Association Studies (GWAS). Our primary objectives were to evaluate existing binary database formats, explore methods for storing and querying positional genomic data, and enhance query throughput through data compression and parallelization.

We developed ZygosDB, a specialized database that outperforms the widely-used Tabix tool, achieving a query throughput speedup of approximately 2-5 times. This significant improvement is attributed to our implementation of efficient data structures, compression techniques, and multi-threaded query execution.

Our findings reveal that while increasing the number of threads initially boosts query throughput, it also introduces overhead that leads to diminished returns beyond a certain point. Additionally, compression algorithms such as Gzip and LZ4 can play a role in optimizing query performance.

Despite these advancements, several open questions and potential improvements remain. Future work should focus on measuring query latency, optimizing memory usage, and further refining the database using profiling tools. Additionally, exploring more specialized columns and enhancing support for rows with start and end positions could yield further performance gains.

Overall, ZygosDB is a solution for efficiently querying large genomic datasets, helping with more effective GWAS.

References

[1] N Tesi, S van der Lee, M Hulsman, H Holstege, and M J T Reinders, “snpXplorer: A web application to explore human snp-associations and annotate snp-sets,” *Nucleic Acids Research* vol. 49, W603–W612, 2021.

[2] H Li, “Tabix: Fast retrieval of sequence features from generic TAB-delimited files,” en, *Bioinformatics* vol. 27, no. 5, pp. 718–719, Jan. 2011.

[3] W J Kent, A S Zweig, G Barber, A S Hinrichs, and D Karolchik, “BigWig and BigBed: Enabling browsing of large distributed datasets,” en, *Bioinformatics* vol. 26, no. 17, pp. 2204–2207, Jul. 2010.

[4] K P Gaffney, M Prammer, L Brasfield, D R Hipp, D Kennedy, and J M Patel, “Sqlite: Past, present, and future,” *Proc. VLDB Endow.* vol. 15, no. 12, pp. 3535–3547, Aug. 2022, ISSN: 2150-8097. DOI: 10.14778/3554821.3554842. [Online]. Available: https://doi.org/10.14778/3554821.3554842.

[5] D R Hipp. “Sqlite is serverless.” (n.d.) [Online]. Available: https://www.sqlite.org/serverless.html (visited on 05/24/2024)

[6] “Rust.” (n.d.) [Online]. Available: https://www.rust-lang.org/ (visited on 05/24/2024)

[7] “Webassembly.” (n.d.) [Online]. Available: https://www.rust-lang.org/what/wasm (visited on 05/24/2024)

[8] D P Wightman, I E Jansen, J E Savage, *et al.*, “A genome-wide association study with 1,126,563 individuals identifies new risk loci for alzheimer’s disease,” en, *Nat Genet* vol. 53, no. 9, pp. 1276–1282, Sep. 2021.

[9] D Kennedy. “Database file format.” (n.d.) [Online]. Available: https://www.sqlite.org/fileformat.html#b_tree_pages (visited on 06/12/2024)

[10] D Knuth, *Sorting and Searching* (The Art of Computer Programming) Second. Addison-Wesley, 1998, vol. 3, pp. 471–479, ISBN: 0-201-89685-0.