GraphMana: graph-native data management for population genomics projects

Ehsan Estaji[1†], Shi-Wei Zhao[1†], Zhao-Yang Chen[1], Shuai Nie[2], Jian-Feng Mao[1*]

[1]Umeå Plant Science Centre, Department of Plant Physiology, Umeå University, SE-90187, Umeå, Sweden.
[2]Rice Research Institute, Key Laboratory of Genetics and Breeding of High Quality Rice in Southern China (Co-construction by Ministry and Province), Ministry of Agriculture and Rural Affairs, Guangdong Key Laboratory of New Technology in Rice Breeding, Guangdong Academy of Agricultural Sciences, 510640, Guangzhou, China.

*Corresponding author(s). E-mail(s): jianfeng.mao@umu.se;
Contributing authors: ehsan.estaji@umu.se; shi-wei.zhao@umu.se; zhao-yang.chen@umu.se; nieshuai@gdaas.cn;
†These authors contributed equally to this work.

Abstract

Population genomics projects rely on fragmented file-based workflows that lose provenance and require full reprocessing when samples are added. GraphMana stores variant data in a graph database as packed genotype arrays with pre-computed population statistics, enabling incremental sample addition, provenance tracking, cohort management, and export to 17 formats. On the human 1000 Genomes Project (3,202 samples, 70.7 million variants), GraphMana completed a 46-operation project lifecycle in 98 minutes from a single persistent database, replacing the ad hoc scripting otherwise required across multiple disconnected tools.

Keywords: population genomics, data management, graph database, variant call format, genotype storage, reproducibility

Population genomics projects at the scale of hundreds to tens of thousands of samples face a data management problem that existing tools do not address. Adding new sequencing batches forces regeneration of every downstream file—every VCF [9], PLINK binary [5], TreeMix input [12], and site frequency spectrum [10]—because flat-file formats encode the complete sample set and cannot be extended in place. Format conversions require bespoke scripts with unrecorded parameters. Annotation updates rewrite entire files despite unchanged genotype data. Provenance is reconstructed forensically from directory timestamps. This coordination burden has become the dominant time sink for mid-scale projects, falling in a gap between single-investigator work (where manual tracking suffices) and biobank-scale programs (where Hail [4] provides managed infrastructure).

The problem is growing. The UK Biobank, All of Us, and TOPMed have collectively sequenced over one million human genomes; the 3,000 Rice Genomes and 1000 Bull Genomes projects exemplify similar expansions in agriculture and livestock; and the Earth BioGenome Project targets 1.67 million species. Across these efforts, sample counts increase continuously, analytical scope broadens, and multi-format data sharing intensifies—yet the underlying file-based infrastructure remains unchanged.

These coordination failures compound through the weekly rhythm of a working project (Supplementary Figs. 1 and 2). A sequencing core delivers 200 new samples, triggering regeneration of every VCF, PLINK binary, and allele frequency table. A collaborator requests an EIGENSTRAT [13] file for a specific population subset, filtered by MAF; producing it requires a one-off script whose parameters go unrecorded. A ClinVar update arrives, but because annotations live inside the VCF header, incorporating new functional data means rewriting the entire genotype file. When questions arise about which samples or filters produced a particular result shared months earlier, the only recourse is to reconstruct the answer from directory timestamps and notebook entries. None of these tasks is computationally hard—bcftools [6] merges files in minutes, PLINK recodes in seconds—but the accumulation of untracked conversions, orphaned scripts, and stale file copies creates a coordination overhead that grows with each new collaborator, annotation source, and export format.

The root cause is structural: population genomics data is inherently relational—variants belong to chromosomes, samples to populations, genes carry functional annotations with versioned provenance—but flat files flatten these relationships into disconnected tables. Several architectures could address this (Table 1). Relational databases require JOIN operations and lack native support for extensible byte-array properties. Columnar stores and Hail excel at matrix computation but treat annotations and provenance as external metadata. Graph databases store data as *nodes* connected by typed *edges*, and queries traverse these connections directly, making lifecycle operations—incremental addition, annotation versioning, cohort definition, provenance binding—persistent and local rather than requiring full-dataset regeneration. We note that graph databases are not the only viable approach; the graph-native model provides a particularly natural fit for the relationship-rich structure of genomic projects.

GraphMana implements this graph-native approach as a persistent database that serves as the project's analytical record (Fig. 1a; Supplementary Fig. 3). Each biallelic

variant is a node carrying a packed genotype array (2 bits per sample, 4 per byte) alongside pre-computed population-level allele counts, sample counts, and frequency arrays (Fig. 1b). This encoding reduces storage 125-fold compared with per-sample edge representations (Supplementary Table 1) and creates a two-tier access model. Most population genetics queries—TreeMix [12], site frequency spectra [10, 11], allele frequency tables—need only population-level aggregates, which are K-element arrays constant in size regardless of sample count; GraphMana serves these via a FAST PATH in $O(K)$ time (Fig. 1c; Supplementary Fig. 4). Per-sample formats (VCF, PLINK [5], EIGENSTRAT [13], Beagle [14], STRUCTURE [15], and 11 others) use a FULL PATH that unpacks genotypes in $O(N)$ time (Supplementary Fig. 5). The practical consequence is that as projects grow from 1,000 to 50,000 samples, the most common analytical exports remain instantaneous (Supplementary Table 2). In total, 17 formats are supported (Supplementary Note 1), of which 6 have been validated against downstream tools and 6 against format specifications (Supplementary Tables 3–4). Genotype roundtrip fidelity exceeds 99.999% (Supplementary Tables 5–6).

Because relationships are explicit graph edges rather than columns embedded in flat files, annotations can be updated by modifying edge properties without touching genotype data—achieving a 27-fold speedup over full VCF rewrites (3.5 s versus 96 s for 53,000 regulatory regions; Fig. 2b). Cohorts are defined as graph queries rather than file extractions. Every export generates a machine-readable manifest recording the software version, filters, and sample set, so that provenance questions are answered by querying metadata rather than reconstructing it from file timestamps (Supplementary Fig. 6).

Incremental sample addition extends packed arrays without modifying existing data. On the human 1000 Genomes Project [3] (70.7 million variants, 3,202 whole-genome samples), adding 234 samples took 182 minutes via a CSV-to-CSV rebuild; approximately 95% of variants required only zero-byte extension (Supplementary Table 7). No downstream result was invalidated because all queries operate on the live database.

We benchmarked GraphMana against bcftools [6] on chromosome 22 (1.07 million variants, 3,202 samples) across five tasks (Supplementary Tables 8–10), with whole-genome export timings reported separately (Supplementary Table 11). GraphMana completed all 46 lifecycle operations in 98 minutes; bcftools completed 17 of 26 in 17 minutes, with 9 operations—multi-format export, in-place annotation, cohort management—having no equivalent (Fig. 2a). For shared operations, bcftools was 3–5× faster per task (Fig. 2b), reflecting sequential file-streaming efficiency versus graph query overhead. The difference is architectural: bcftools optimizes individual file operations, whereas GraphMana eliminates the repeated regeneration and manual coordination between them.

GraphMana currently represents variants as biallelic nodes; multi-allelic records are decomposed at import and reconstructed during VCF export (Supplementary Note 2). Structural variants are stored with typed properties. gVCF reference blocks are imported as variant nodes and can be filtered with `--filter-variant-type`. The platform operates as a single-writer system with multi-reader support via replicas.

Packed encoding scales linearly: $\lceil N/4 \rceil$ bytes per variant for genotypes, $\lceil N/8 \rceil$ for phase. All scaling estimates assume human whole-genome data (∼85M biallelic variants); for other species or designs, requirements scale proportionally by variant count (Supplementary Tables 1–2). At 100–10,000 samples, all operations are interactive; beyond 50,000 samples, the single-node architecture becomes a bottleneck, and distributed frameworks [4] are more appropriate. The current implementation uses Neo4j Community Edition [2]; the design is not vendor-specific. The companion GraphPop engine [1] provides graph-native analytical computation on the same database.

File-based workflows are inherently ephemeral: each computation produces an output that may or may not be retained, with no systematic connection between the data, the analysis, and the parameters that produced it. GraphMana replaces this with a persistent analytical record—a single queryable database in which genotype data, population statistics, annotations, cohort definitions, and provenance coexist and evolve together. As population genomics projects grow in duration, scale, and collaborative complexity, this shift from ephemeral files to persistent project state addresses a reproducibility gap that individual tool improvements cannot close. Software, documentation (11 vignettes, 65 command reference pages), and pre-built databases are available under the MIT license.

Online Methods

Graph database technology

A graph database stores data as nodes (entities) and edges (typed, directed relationships), with key-value properties on both. Unlike relational databases, which reconstruct relationships through table joins at query time, graph databases traverse pre-materialized connections in constant time per hop—a property known as index-free adjacency. This distinction matters for genomic data because the relationships between variants, chromosomes, genes, populations, and annotation sources are numerous, typed, and frequently traversed.

In GraphMana's property-graph model, a Variant node connects to its Chromosome via an ON_CHROMOSOME edge, to Gene nodes via HAS_CONSEQUENCE edges carrying annotation properties (consequence type, impact level, annotation version), and to the ordered variant chain via NEXT edges. Sample nodes connect to Population nodes via IN_POPULATION edges. Updating an annotation version modifies HAS_CONSEQUENCE edge properties without touching genotype arrays; defining a cohort is a graph traversal that produces a sample list without extracting files; and provenance is recorded as IngestionLog nodes linked to the operations that produced them. These operations are local to the edges or nodes involved, not to the entire dataset.

We chose Neo4j Community Edition [2] for its mature property-graph implementation, the Cypher query language, ACID transaction support, and free open-source licensing. The graph-native design principles—typed nodes and edges, byte-array properties on nodes, edge-independent updates—are not specific to Neo4j and could be

implemented on other property-graph databases. A concrete end-to-end project workflow is presented in Supplementary Note 3, and Supplementary Note 4 provides a detailed introduction for readers unfamiliar with graph database concepts.

Software architecture

GraphMana comprises a Python command-line application (21,267 lines of source code) with a bundled Java plugin (2,864 lines) for server-side Neo4j procedures. The Python layer handles VCF parsing via cyvcf2 [8], genotype packing and unpacking via NumPy (complementing scikit-allel [7] for lower-level variant access), CSV generation for bulk database loading, and all 17 export format writers. The Java plugin provides optional server-side packed array extension for incremental imports, avoiding Bolt protocol round-trips for large batch operations.

The 58 CLI commands are organized into nine functional domains: data import and integration (5 commands), annotation management (9), data export (2), sample and cohort management (11), quality control and verification (3), provenance and state tracking (7), database administration (8), status and reporting (5), and infrastructure and deployment (6). The full command hierarchy is shown in Supplementary Fig. 7. A companion Python package (graphmana-py) provides a Jupyter-compatible API returning pandas DataFrames for interactive analysis. The test suite comprises 1,439 unit and integration tests covering genotype packing, phase conventions, byte boundary conditions, export format contracts, filter chains, and database operations.

Data encoding

The graph database contains five primary node types—Variant, Sample, Population, Chromosome, and Gene—connected by four relationship types: ON_CHROMOSOME, IN_POPULATION, HAS_CONSEQUENCE, and NEXT (chromosomal ordering). Variant nodes store genotype data as packed byte arrays using a 2-bit encoding: gt_packed stores diploid genotypes (00 = homozygous reference, 01 = heterozygous, 10 = homozygous alternate, 11 = missing), yielding 4 genotypes per byte. Phase information is stored separately in phase_packed (1 bit per sample, 8 per byte), and ploidy status in ploidy_packed (1 bit per sample; null when all samples are diploid). Genotype values are remapped from cyvcf2's internal coding (where homozygous alternate = 3 and missing = 2) to the sequential 2-bit encoding at import time; the remap is applied exactly once per code path and verified by 33 dedicated unit tests. This packed representation achieves a 125-fold storage reduction compared with an alternative design using individual Sample-to-Variant edges (Supplementary Table 1).

Each Variant node also carries K-element population-level pre-computed arrays: pop_ids (population identifiers), ac (allele counts per population), an (allele numbers), af (allele frequencies), het_count, hom_alt_count, and het_exp (expected heterozygosity under Hardy–Weinberg equilibrium). These arrays have K elements where K is the number of populations (typically 5–30), making them constant in size regardless of sample count N. This property enables the FAST PATH access pattern: queries that need only population-level statistics (TreeMix, SFS, allele frequency tables) read these pre-computed arrays directly without unpacking any per-sample genotype data. Full

encoding details and a worked multi-allelic decomposition example are provided in Supplementary Note 2.

Import pipeline

GraphMana uses a two-step split pipeline designed for both workstations and HPC clusters. The first step (prepare-csv) parses VCF files using cyvcf2 for streaming access, packs genotypes into byte arrays, computes population statistics from the genotype calls, and writes Neo4j-compatible CSV files. This step is embarrassingly parallel across chromosomes and requires no running database instance, making it suitable for batch job submission on compute nodes without Neo4j. The second step (load-csv) uses neo4j-admin import for bulk loading, which writes directly to the Neo4j store files and bypasses the transaction engine, achieving import speeds limited only by disk I/O.

On our benchmark hardware (32 CPU cores, 64 GB RAM, NVMe SSD), CSV generation for 2,500 human whole-genome samples across 22 chromosomes (70.7 million variants) completed in 95 minutes with 16 parallel threads; bulk loading of the resulting 214 GB of CSV files completed in 3 minutes.

Incremental sample addition

Adding new samples to an existing database extends the packed genotype arrays of all existing Variant nodes. GraphMana provides three strategies of increasing throughput: (1) Cypher transactions on a running database instance, suitable for small additions (fewer than 10,000 variants); (2) an export-extend-reimport path that reads variant data from Neo4j, extends arrays in Python, and reimports via neo4j-admin; and (3) a CSV-to-CSV rebuild that reads the existing CSV checkpoint directly at NVMe speed, extends arrays using NumPy vectorized operations, and produces a new CSV set for reimport. The CSV-to-CSV path is fastest for whole-genome databases: on our benchmark dataset, extending 70.7 million variants with 234 new samples completed in 182 minutes, of which 160 minutes was sequential CSV I/O and array extension, 15 minutes was neo4j-admin import, and 7 minutes was VCF parsing and database restart. Approximately 95% of variants (69.6 million of 70.7 million) were on chromosomes absent from the new VCF and required only HomRef extension—appending zero bytes to packed arrays without unpacking and repacking.

Correctness validation

We validated genotype fidelity by importing human 1000 Genomes Project chromosome 22 data (897,645 biallelic SNPs, 5 samples), exporting to phased VCF, and comparing per-sample genotypes with the original using bcftools. Concordance exceeded 99.999%, with 2–8 mismatches per sample confined to multi-allelic positions where position-based joining during comparison could not distinguish co-located biallelic records. A systematic correctness matrix covering biallelic SNPs, indels, multi-allelic sites, structural variants, phased and unphased genotypes, missing data, haploid and diploid ploidy, and annotation-layer independence is provided in Supplementary Tables 5–6.

Benchmark methodology

All benchmarks used the human 1000 Genomes Project high-coverage dataset [3] (GRCh38). Chromosome 22 (1,066,557 variants, 3,202 samples, 26 populations) served as the primary target for the full comparison matrix. The 3,202 samples were split deterministically (seed = 42) into a base cohort of 2,500 and three incremental batches of 234 each.

We compared GraphMana against bcftools [6] (version 1.17) as the single comparison target, chosen as the stronger baseline for VCF-centric operations. Five benchmarks were conducted: (1) incremental sample addition (GraphMana CSV-to-CSV rebuild versus bcftools merge, 3 rounds); (2) cohort-specific VCF extraction (5 superpopulation cohorts); (3) multi-format export (6 formats from a single source; bcftools supports VCF only); (4) annotation update (53,000 BED regulatory regions; in-place edge modification versus VCF rewrite); and (5) a lifecycle simulation comprising 7 phases of interleaved imports, exports, and annotation updates. All timings report wall-clock time on a workstation with 32 CPU cores, 64 GB RAM, and NVMe SSD. The graph database was configured with 16 GB heap and 16 GB page cache. Each benchmark was run twice; reported values are from the second run (warm page cache). We emphasize that per-operation timings favor bcftools, which is optimized for sequential file streaming; GraphMana's advantage lies in avoiding repeated file regeneration across the project lifecycle. Benchmark scripts and raw timing data are deposited at Zenodo.

Scalability

Packed encoding scales linearly: storage per variant is $\lceil N/4 \rceil$ bytes for genotypes and $\lceil N/8 \rceil$ bytes for phase, where N is the number of samples. All scaling estimates in this work assume human whole-genome data ($\sim$85 million biallelic variants per genome); for species or designs with fewer variants (exome sequencing at $\sim$5M variants, rice at $\sim$29M), storage and time requirements scale proportionally by variant count. At 100–10,000 human whole-genome samples, all operations are interactive. At 10,000–50,000 samples, FAST PATH operations remain instantaneous because population arrays are constant size regardless of N, but FULL PATH exports scale linearly and become the bottleneck. Beyond 50,000 samples, the single-node graph database architecture imposes practical limits due to per-variant query latency and transaction log pressure. At biobank scale (500,000+ samples), distributed computing frameworks such as Hail [4] are more appropriate. Supplementary Tables 1–2 provide detailed storage and performance estimates from 3,202 to 500,000 samples.

Deployment

GraphMana installs entirely in user space without administrator privileges. The primary installation method is `pip install graphmana`, which bundles the pre-built Java procedures JAR as package data. The `graphmana setup-neo4j` command downloads and configures Neo4j Community Edition, automatically deploys the JAR to the plugins directory, and optionally downloads Eclipse Temurin JDK 21 via the `--install-java` flag for systems without Java. For HPC clusters, a two-step split

pipeline separates CSV generation (which runs on any compute node without Neo4j) from bulk loading (which requires the database host). SLURM and PBS job scripts are provided. The Neo4j data directory must reside on local SSD or scratch storage; network filesystems (NFS, Lustre, GPFS) cause severe performance degradation, and GraphMana includes a filesystem check command that warns users before import.

Data availability. GraphMana is available at https://github.com/jfmao/GraphMana under the MIT license. Benchmark data and pre-built databases are deposited at Zenodo (https://doi.org/10.5281/zenodo.19472836).

Code availability. All source code is at https://github.com/jfmao/GraphMana under the MIT license.

Acknowledgements. This work was partially supported by the Wallenberg Initiatives in Forest Research (WIFORCE) funded by the Knut and Alice Wallenberg Foundation. Genomic data processing and analyses were performed using resources provided by the Swedish National Infrastructure for Computing (SNIC), through the High Performance Computing Centre North (HPC2N) at Umeå University. The human 1000 Genomes Project dataset [3] was used under its open data access terms.

Author contributions. Conceptualization and design: J.-F.M., E.E., and S.N. Implementation and testing: E.E., S.-W.Z., Z.-Y.C., and S.N. Writing and editing: J.-F.M., E.E., and S.-W.Z.

Competing interests. The authors declare no competing interests.

References

[1] Estaji, E. et al. GraphPop: graph-native computation decouples population genomics complexity from sample count. https://github.com/jfmao/GraphPop (companion paper, in preparation).

[2] Neo4j, Inc. Neo4j Graph Database. https://neo4j.com (Community Edition 5.x).

[3] Byrska-Bishop, M. et al. High-coverage whole-genome sequencing of the expanded 1000 Genomes Project cohort including 602 trios. *Cell* **185**, 3426–3440.e19 (2022).

[4] Hail Team. Hail: An Open-Source Framework for Scalable Genomic Data Analysis. https://hail.is.

[5] Chang, C.C. et al. Second-generation PLINK: rising to the challenge of larger and richer datasets. *GigaScience* **4**, s13742-015-0047-8 (2015).

[6] Danecek, P. et al. Twelve years of SAMtools and BCFtools. *GigaScience* **10**, giab008 (2021).

[7] Miles, A. et al. scikit-allel: A Python package for exploring and analysing genetic variation data. https://github.com/cggh/scikit-allel.

[8] Pedersen, B.S. & Quinlan, A.R. cyvcf2: fast, flexible variant analysis with Python. *Bioinformatics* **33**, 1867–1869 (2017).

[9] Danecek, P. et al. The variant call format and VCFtools. *Bioinformatics* **27**, 2156–2158 (2011).

[10] Gutenkunst, R.N. et al. Inferring the joint demographic history of multiple populations from multidimensional SNP frequency data. *PLoS Genetics* **5**, e1000695 (2009).

[11] Excoffier, L. et al. fastsimcoal2: demographic inference under complex evolutionary scenarios. *Bioinformatics* **37**, 4882–4885 (2021).

[12] Pickrell, J.K. & Pritchard, J.K. Inference of population splits and mixtures from genome-wide allele frequency data. *PLoS Genetics* **8**, e1002967 (2012).

[13] Patterson, N., Price, A.L. & Reich, D. Population structure and eigenanalysis. *PLoS Genetics* **2**, e190 (2006).

[14] Browning, S.R. & Browning, B.L. Rapid and accurate haplotype phasing and missing-data inference for whole-genome association studies by use of localized haplotype clustering. *Am. J. Hum. Genet.* **81**, 1084–1097 (2007).

[15] Pritchard, J.K., Stephens, M. & Donnelly, P. Inference of population structure using multilocus genotype data. *Genetics* **155**, 945–959 (2000).

Figure Legends

Figure 1. From VCF to graph database to 17 export formats. (**a**) Genetic variants in a flat file (Variant Call Format, VCF), with columns color-coded by their destination in the graph—chromosome (gold), variant identity (blue), sample identifiers (pink), and per-sample genotype calls (green). (**b**) After `graphmana ingest`, the same genetic variants are stored in a graph database: Variant nodes are linked in genomic order by NEXT relationships, attached to Chromosome nodes via ON_CHROMOSOME, and associated with Sample, Population, and Gene nodes via typed edges. Each Variant node stores two data representations side by side, one feeding each of the two access paths shown in panel c. Left (purple, FULL PATH source): per-sample genotypes encoded as `gt_packed` (2 bits per sample, 801 bytes for 3,202 samples) and `phase_packed` (1 bit per sample), which grow linearly with N. Right (green, FAST PATH source): pre-computed population arrays (`pop_ids[]`, `ac[]`, `af[]`, `an[]`, `het_count[]`, `hom_alt_count[]`) containing K elements (K = number of populations), constant in size regardless of N. This dual representation is the architectural foundation for the two export paths. (**c**) The `graphmana export` command selects one of two access paths based on the requested format. FULL PATH (purple, $O(N)$ per variant) unpacks `gt_packed` and `phase_packed` and translates to 12 format-specific encodings (VCF, PLINK, EIGENSTRAT, Beagle, STRUCTURE, and seven others); used for PCA, admixture, GWAS, phasing, imputation, and relatedness (42 s for PLINK export of chr22, 3,202 samples). FAST PATH (green, $O(K)$ per variant) reads `ac[]`, `an[]`, `af[]` directly, skipping `gt_packed`; used for population tree inference, demographic modeling, and frequency queries via 5 formats (TreeMix, SFS-dadi, SFS-fsc2, BED, TSV; 0.6 s for TreeMix on the same dataset, a 70× speedup). FAST PATH runtime is independent of sample count.

Figure 2. Human 1000 Genomes Project benchmark (chromosome 22, 1.07 million variants, 3,202 samples). (**a**) Lifecycle simulation across 46 operations spanning import, filtering, cohort extraction, annotation update, and multi-format export. GraphMana completes all 46 operations from a single persistent database; bcftools completes 17 of 26 attempted operations, with 9 operations (multi-format export, in-place annotation update, cohort management) having no equivalent in a flat-file workflow and shown in grey. (**b**) Per-task timing comparison for the operations supported by both tools. Annotation updates are 27× faster in GraphMana (3.5 s versus 96 s for 53,000 regulatory regions) via in-place edge modification. Multi-format export covers 6 formats (VCF, PLINK, EIGENSTRAT, TreeMix, SFS, BED) versus VCF only for bcftools. Per-task timings favor bcftools for shared file-streaming operations (3–5× faster); GraphMana's advantage emerges over the project lifecycle by eliminating repeated file regeneration and manual coordination.

Table 1. Data management architectures for population genomics. Operations natively supported (•), supported only with external scripting (○), or not supported (—). The four columns compare flat files (VCF/BCF, PLINK, EIGENSTRAT, etc.), relational databases, columnar/Hail-style frameworks, and GraphMana.

GraphMana is the only architecture that natively supports all six lifecycle operations: incremental sample addition, annotation versioning, provenance binding, cohort-as-query, multi-format export, and in-place annotation update.

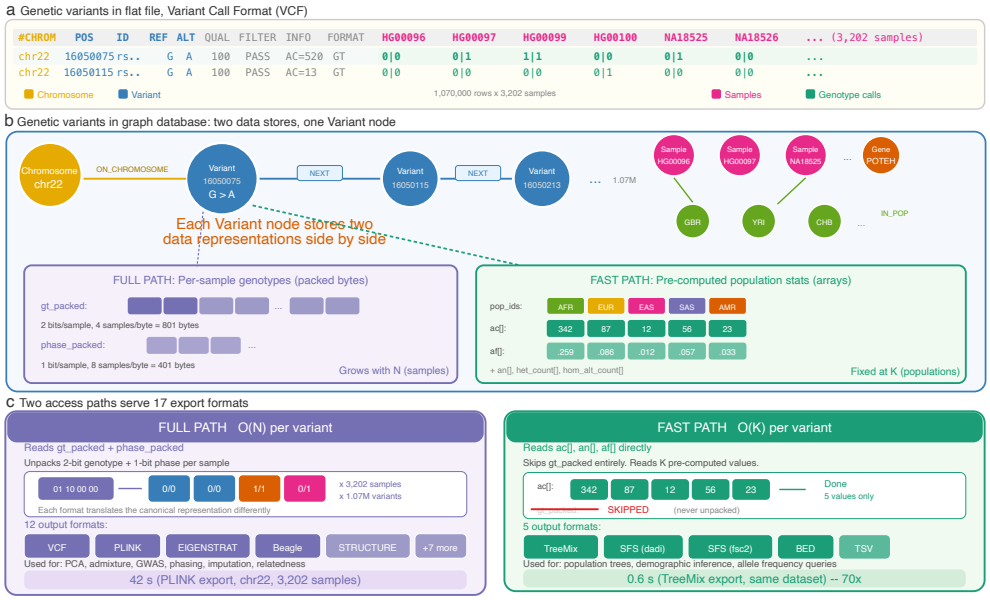

Fig. 1 From VCF to graph database to 17 export formats.

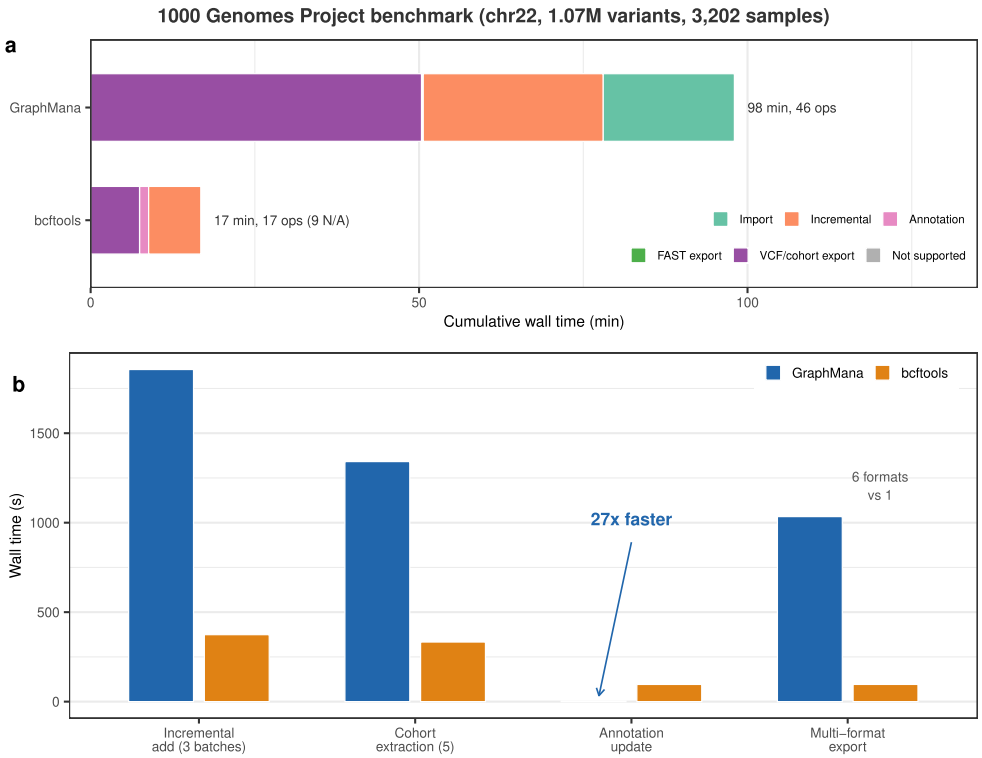

Fig. 2 Human 1000 Genomes Project benchmark.

Table 1 Data management architectures for population genomics.

Operation	Flat files	Relational DB	Columnar/Hail	GraphMana
Incremental sample addition	—	○	○	●
Annotation versioning	—	○	—	●
Provenance binding	—	○	○	●
Cohort as query	—	●	●	●
Multi-format export (17)	○	—	○	●
In-place annotation update	—	●	—	●