SQLite: Grand Unified Genomics File Format?

github.com/mlin/GenomicSQLite

Michael F. Lin, PhD
@DNAmlin
dna@mlin.net

Introduction

Genomic data models often materialize as bespoke, binary file formats, historically owing to our field's demanding performance & compression needs, and our inclination to neutrality amongst stacks/platforms/vendors.

We revisit SQL as a primary medium for defining genomic data models. Perhaps SQLite -- with a few key features added -- can serve as a portable file format for *any* data model, better enabling interoperation with the "big data" ecosystem.

What is SQLite?

SQLite is a lightweight relational database manager supporting full-featured SQL operations. It's an efficient local storage engine because it doesn't incur network requests.

The public-domain C language library is accessible to all modern platforms and languages, and indeed serves aboard almost all computing devices today.

A database is stored in one file, inside which many SQL tables and indexes can coexist. Unlike text formats, it intrinsically supports indexed access and in-place update transactions.

Individual databases may grow to many terabytes, supported by lesser-known scalability features:

+ Automatic paging between storage & memory
+ Access from many threads
+ Parallel external sorter

Notable too: raw BLOB storage; JSON columns & indexes; recursive SQL algorithms.

Genomics Extension for SQLite

The new features are packaged in the Genomics Extension for SQLite ("GenomicSQLite", Apache licensed). After a special invocation to open the database, programmers use it just like SQLite.

```
import sqlite3
import genomicsqlite

dbconn: sqlite3.Connection =
genomicsqlite.connect("compressed.db")
```

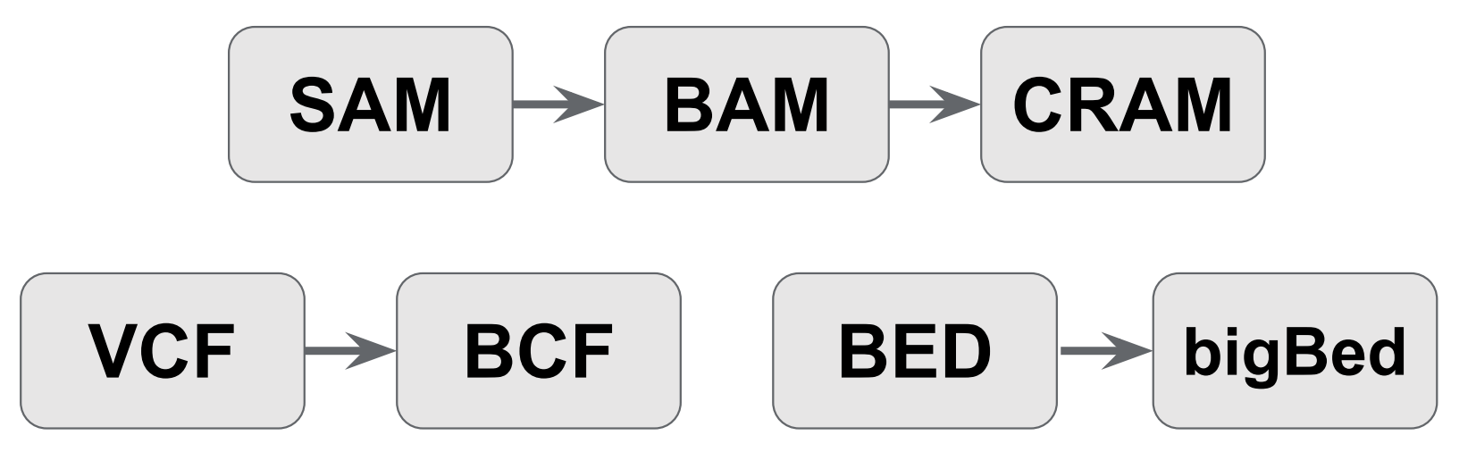
Language bindings for Python, Rust, Java/JVM, C/C++, and are easy to add. (Help wanted!)

Programming Guide: mlin.github.io/GenomicSQLite

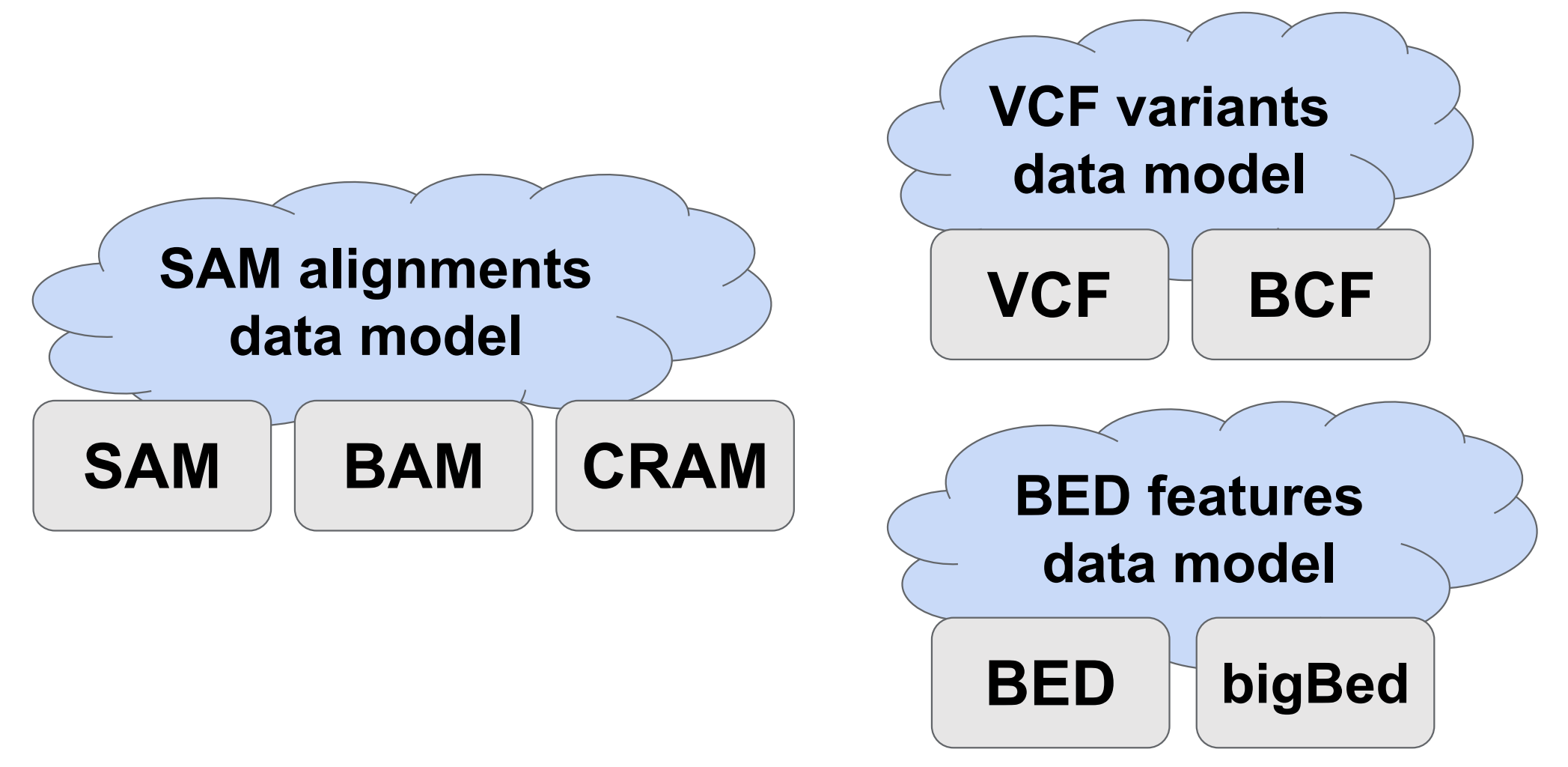
Data Models vs. File Formats

Bioinformatics & genomics have long preferred minimalistic text file formats for data exchange, which are accessible & portable, but also brittle & inefficient.

The production data volumes of next-generation sequencing (NGS) motivated compressed binary equivalents of initial text formats.



These examples illustrate how abstract data models can materialize in multiple file formats.



But when the data models are *conceived as* file formats, they inherit prototypical constraints. They prefer line-oriented 1D/2D concepts, they're unindexed & read-only, and they tend to grow by grafting new information into free text fields.

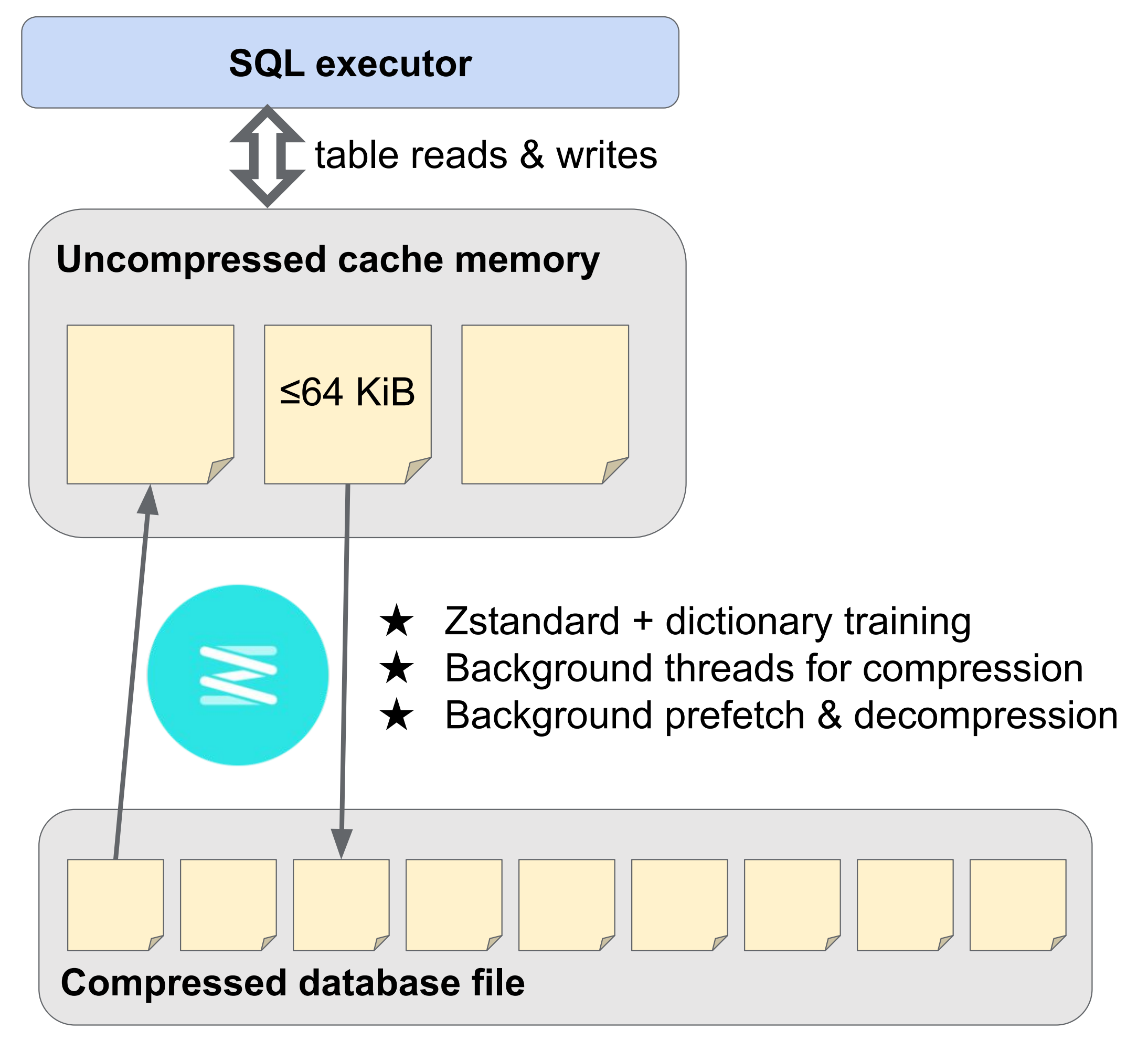
Querying them and integrating them with *other* data types are custom operations, leading to extensive duplication of effort across the field.

Can we uncouple data models from file formats without sacrificing portability & efficiency?

Database compression

SQLite's lack of built-in compression was often uneconomical for bioinformatics data storage. (Zipped database files would still need to be unzipped to access or update.)

We added a general-purpose, read/write compression layer to SQLite's existing file pagination scheme. This applies to any database schema, totally transparent to SQL operations.

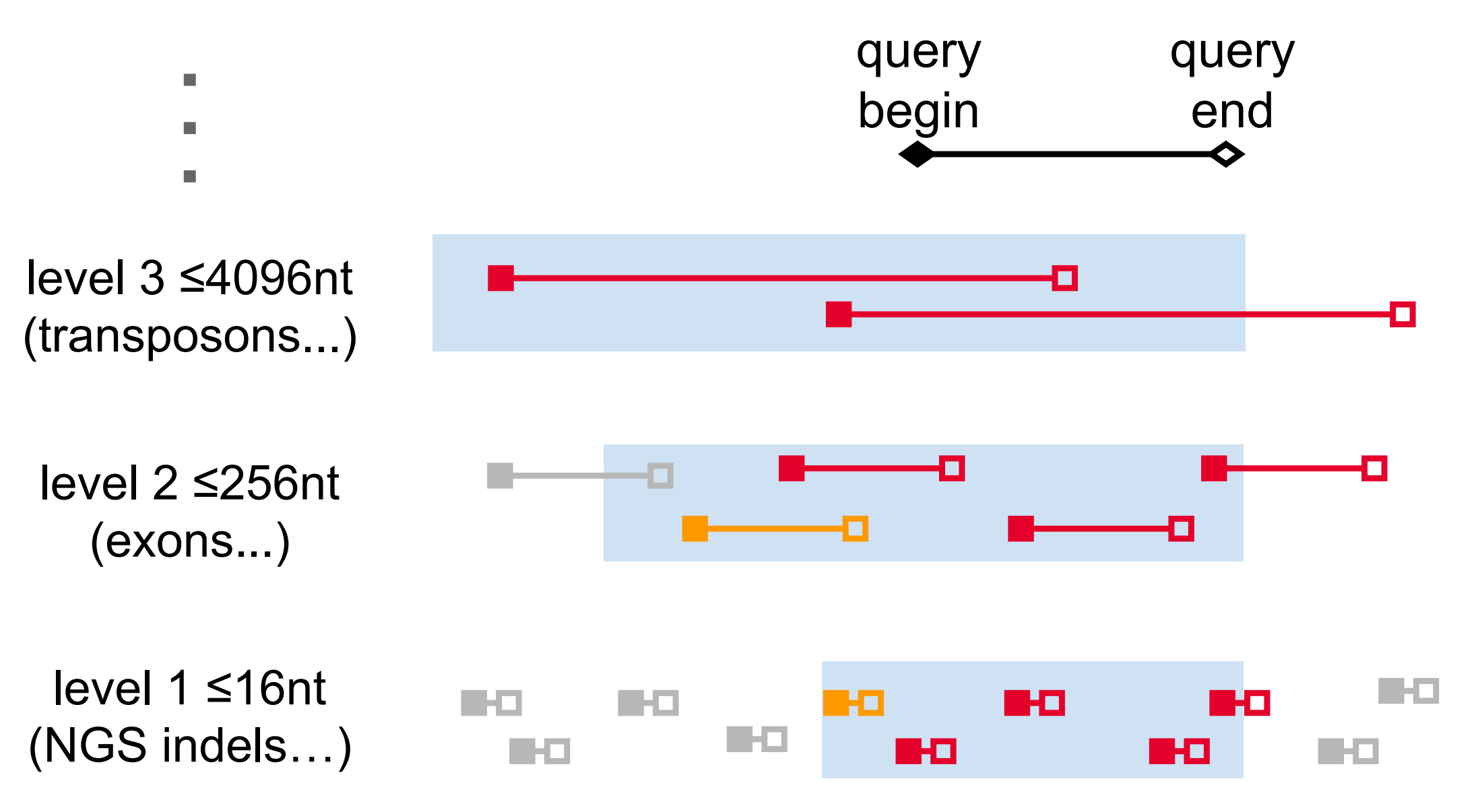

SQL executor
table reads & writes
Uncompressed cache memory
≤64 KiB
★ Zstandard + dictionary training
★ Background threads for compression
★ Background prefetch & decompression
Compressed database file

This recalls BAM file compression (BGZF) but with transactional updates and the speedier Zstandard. Future work can further explore archival database storage with columnar & reference-based compression.

Genomic Range Indexing

Genomic datasets are frequently queried by genomic range, e.g. for all features overlapping chr2:100,123,000-100,456,000

SQL indexes aren't immediately suitable for such queries, especially for lengthy feature types. We added a genomic range index (GRI) to SQLite.


query begin
query end
level 3 ≤4096nt (transposons...)
level 2 ≤256nt (exons...)
level 1 ≤16nt (NGS indels...)

The feature index is partitioned by length magnitude. Features on each "level" are searched by begin position in the query range, extended upstream by the level max length (blue background). Higher levels search longer ranges, but only for features of comparable length. Finally, the multi-level results are unioned.

The technique recalls schemes found in tabix, UCSC Genome Browser, and Ensembl, but easily applied to any SQLite table with (chromosome, begin, end) columns. Creating GRI in Python:

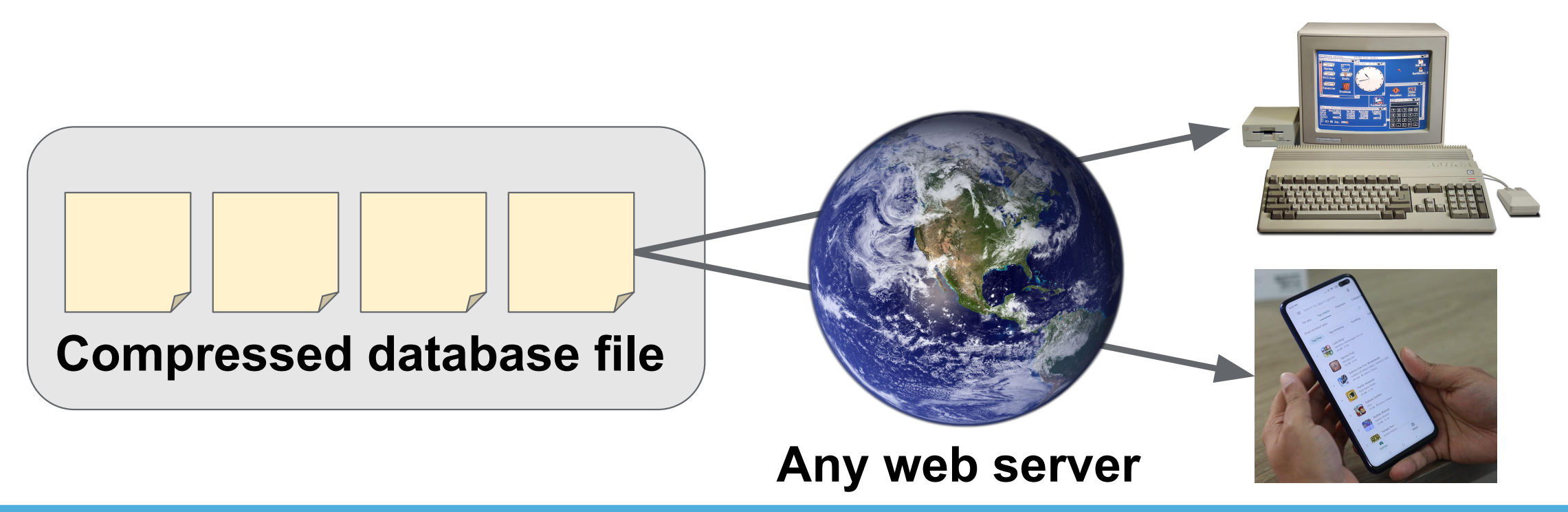
```
dbconn.executescript(
genomicsqlite.create_genomic_range_index_sql(
"myTable", "chromCol", "beginCol", "endCol"))
```

GRI query in SQL:

```
SELECT * FROM myTable WHERE myTable._rowid_
IN genomic_range_rowids("myTable", "chr2",
100123000, 100456000)
```

Web access & query

We added the ability to read SQLite databases over the web for indexed remote access. Queries transfer only the necessary compressed pages of the database file, using HTTPS byte range requests (no special server software needed).



Are we there yet?

Moreso than the prior decade, rapid innovation in sequencing tech & their informatics techniques look set to demand scalable standards for numerous new data models.

A container for *any* schema, with

+ block compression
+ genomic indexing
+ accessibility to all programs (C ABI)
+ web access
+ vendor neutrality

...could reduce duplication of effort -- permitting designers to focus on data model concepts over serialization details.

With self-describing SQL schemas, database files can be replicated into other SQL-based systems easily, without schema-specific conversion logic. This facilitates integration with pheno/clinical data typically stored in relational databases.