GT-Pro Query Engine

Contents

1 Database Size Reduction

GT-Pro's gut genome database consists of 2.8 billion kmers that cover 52.8 million bi-allelic SNPs across 881 species. Each kmer is 31 base pairs long and occurs in a unique species.

GT-Pro uses 4 bytes to represent each kmer and 48 bytes to represent each SNP, for a total of 13 GB – twice as compact compared to bzip2 compression.

- The database download takes less time.

- Queries run at full speed immediately after download.

- Parallel runs share the database in RAM via memory mapping.

To accelerate queries, GTPro uses an index and a bloom filter for the kmers present in the database. These auxiliary data structures may be downloaded with the database or built locally from it.

In AWS, all required downloads – of the database, index, and bloom filter – complete in 30 seconds or less. With slower networking, one has the option to download just the database, then build the index and bloom filter locally.

For faster download speeds in AWS, we recommend using an instance with at least 10 Gbps networking, and following the instructions in `https://github.com/chanzuckerberg/s3mi`.

1.1 Kmers Table

The database consists of two tables.

- A 10.6 GB table containing a 4-byte entry for each kmer.

- A 2.4 GB table containing a 24-byte entry for each allele.
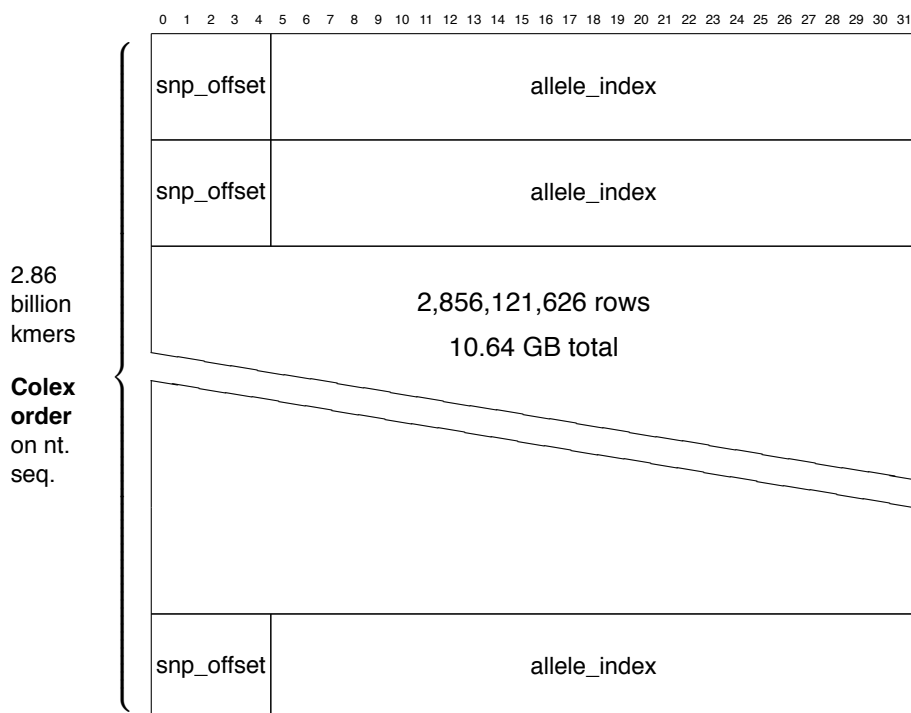
Each entry in the **Kmers Table** is comprised of a 5 bit *snp_offset* and 27 bit *allele_index* that together represent a kmer.

The *snp_offset* identifies the SNP position within the kmer. It's the count of nucleotides in the kmer that precede the SNP, a number between 0 and 30.

The *allele_index* points to an entry in the **Alleles Table** from which the kmer's entire sequence can be recovered through the formula

$$kmer\_seq := \mathrm{substr}(allele\_seq, 30 - snp\_offset, 31)$$

See subsection "Overlap Encoding" below.

1.2 Overlap Encoding

The major and minor allele of a SNP may each be covered by up to 31 different kmers, with *snp_offset* values 0, 1, ... 30. Some kmers would be absent from the database because they bear too little information about their origin (for instance, kmers that occur in multiple species).

In the illustration below, only 6 of the possible 31 kmers are present for the major allele, and 5 for the minor allele. There are two distinct kmers with *snp_offset* = 10. The first of those covers *allele_seq_major* and the second covers *allele_seq_minor*.

```
                                       SNP                                       snp_offset

.............................. G GTCTAGGCGCAATGTAACGCTTTTATCGCT              0
.....................TCTT G GTCTAGGCGCAATGTAACGCTTTTAT....              4
..................GCTTAGTCTT G GTCTAGGCGCAATGTAACGC..........          10'
...............TTAGCGCTTAGTCTT G GTCTAGGCGCAATGT...............      15
............GACTTAGCGCTTAGTCTT G GTCTAGGCGCAA.................      18
........CTAGACTTAGCGCTTAGTCTT G GTCTAGGCG.....................      21
CAACTATTCCTAGACTTAGCGCTTAGTCTT G GTCTAGGCGCAATGTAACGCTTTTATCGCT  allele_seq_major

CAACTATTCCTAGACTTAGCGCTTAGTCTT C GTCTAGGCGCAATGTAACGCTTTTATCGCT  allele_seq_minor
..................GCTTAGTCTT C GTCTAGGCGCAATGTAACGC..........          10"
..............CTTAGCGCTTAGTCTT C GTCTAGGCGCAATG................      16
........CTAGACTTAGCGCTTAGTCTT C GTCTAGGCG.....................      21
.....ATTCCTAGACTTAGCGCTTAGTCTT C GTCTA.........................      25
....TATTCCTAGACTTAGCGCTTAGTCTT C GTCT..........................      26
```

Altogether, the 11 kmers above contain 341 base pairs. Their shared *allele_seq*'s (major and minor) comprise only 122 base pairs, from which the kmers can be recovered fully. The compression ratio is signifficantly higher in the actual database, where the average number of kmers per allele is 28.

Each 61-bp *allele_seq* is stored into a separate entry in the **Alleles Table**.

4

1.2.1 Multi-SNP Kmers

If multiple SNPs occur in close proximity, some of the overlapping kmers may not completely agree. In the illustration below, a lightning symbol indicates the position of a second SNP over which some kmers disagree.

```
        ⚡                    SNP                          snp_offset

...............⚡...............  G  GTCTAGGCGCAATGTAACGCTTTTATCGCT        0
...............⚡...........TCTT  G  GTCTAGGCGCAATGTAACGCTTTTAT....        4
...............⚡.........AGTCTT  G  GTCTAGGCGCAATGTAACGCTTTT......        6
...........GACTTAGCGCTTAGTCTT  G  GTCTAGGCGCAA.................        18'
CAACTATTCCTAGACTTAGCGCTTAGTCTT  G  GTCTAGGCGCAATGTAACGCTTTTATCGCT    allele_seq_major

CAACTATTCCTAGACTTAGCGCTTAGTCTT  C  GTCTAGGCGCAATGTAACGCTTTTATCGCT    allele_seq_minor

...........GACTTAGCGCTTAGTCTT  C  GTCTAGGCGCAA.................        18"
.........CTAGACTTAGCGCTTAGTCTT  C  GTCTAGGCG....................        21
CAACTATTCCTAGAGTTAGCGCTTAGTCTT  C  GTCTAGGCGCAATGTAACGCTTTTATCGCT    allele_seq_minor_2

........CCTAGAGTTAGCGCTTAGTCTT  C  GTCTAGGC.....................        22
....TATTCCTAGAGTTAGCGCTTAGTCTT  C  GTCT........................        26
```

Encoding the discordant kmers in this case requires the introduction of another entry into the **Alleles Table** to represent *allele_seq_minor_2*. In some cases, up to 11 SNPs have been seen in such close proximity. Approximately 1.8 percent of entries in the **Alleles Table** exist for this purpose.

1.3   Alleles Table

| 0 1 | 55 | 61 63 |

```
┌──┬──────────────────────────────────────────────────┬────┐
│XX│        allele_seq: 30 bp before SNP (60 bits)    │SNP │  ⎫
├──┼──────────────────────────────────────────────────┼────┤  │ allele
│SNP│       allele_seq: 30 bp after SNP (60 bits)     │ XX │  ⎬ entry
├───┴─────────────────────────────────────────┬───────┴────┤  │ 0
│          snp_coord (56 bits)                 │  reserved  │  ⎭
└──────────────────────────────────────────────┴───────────┘
```

```
┌──┬──────────────────────────────────────────────────┬────┐
│XX│        allele_seq: 30 bp before SNP (60 bits)    │SNP │  ⎫
├──┼──────────────────────────────────────────────────┼────┤  │ allele
│SNP│       allele_seq: 30 bp after SNP (60 bits)     │ XX │  ⎬ entry
├───┴─────────────────────────────────────────┬───────┴────┤  │ 1
│          snp_coord (56 bits)                 │  reserved  │  ⎭
└──────────────────────────────────────────────┴───────────┘
```

107,565,805 alleles (1.8% multi-SNP)

for 52.8 million SNPs

3 dwords per allele

2.4 GB total

```
┌──┬──────────────────────────────────────────────────┬────┐
│XX│        allele_seq: 30 bp before SNP (60 bits)    │SNP │  ⎫
├──┼──────────────────────────────────────────────────┼────┤  │ allele
│SNP│       allele_seq: 30 bp after SNP (60 bits)     │ XX │  ⎬ entry
├───┴─────────────────────────────────────────┬───────┴────┤  │ N − 1
│          snp_coord (56 bits)                 │  reserved  │  ⎭
└──────────────────────────────────────────────┴───────────┘
```

Each entry in the **Alleles Table** represents a specific SNP and allele by encoding the 61 nucleotide *allele_seq* centered on the SNP, along with the species id (6 decimal digits), major/minor allele indicator (0 or 1), and genomic coordinate of the SNP. The allele nucleotide itself, *allele_seq(30)*, is represented redundantly in dwords 0 and 1.

2 Query Acceleration

2.1 Index

For any nucleotide sequence *S*, the kmers that end with *S* occupy consecutive rows in the **Kmers Table**. They can be located quickly with an index that simply maps *S* to the range of rows that share *S*.

The following algorithm visits every allele in the database that is covered by a given 31-bp kmer *query*.

- Look up 16-bp suffix of *query* in **Index**.

- If found, examine range of entries in **Kmers Table** referred by **Index**.

- Decode each referred kmer entry through **Alleles Table** to recover a *db_kmer* that's a possible match for *query*. Compare entire nucleotide sequence of *db_kmer* to *query*. On exact match, emit allele info.

2.2 Bloom Filter

The index above quickly rejects kmer queries whose suffix is absent from the database. A similar method rejects queries with absent prefix: the set of 18-bp kmer prefixes in the database is encoded as a bit vector and consulted for each query. We call that a bloom filter.

In more detail, a vector of 2^{36} bits (3 GB) is initialized so *bit_vector[bit_address]* is 1 if and only if interpreting *bit_address* itself as an 18-bp nucleotide sequence (see next section) yields the prefix of some kmer in the database.

The amended query algorithm is now

- Look up *query* prefix in **Bloom Filter**.

- If found, look up *query* suffix in **Index**.

- If found, examine all **Kmers Table** entries referred by the index one by one, as before.

Why exactly 16 nucleotide suffix and 18 nucleotide prefix? GT-Pro supports a range of values for these parameters, with defaults that perform best for the available amount of RAM. See figure "Query Speed vs Memory Footprint".

2.3 Integer Encoding

In binary encoding each nucleotide takes 2 bits.

Nucleotide strings are encoded so that their initial characters map to lower memory addresses / less significant bits.

A common trick is to interpret a nucleotide sequence as an integer, use that integer as an array index, or sort those integers to create a substrate for efficient sequence lookups.

The following table illustrates the correspondence between integer sort order on the encodings and **colex order** on the nucleotide strings. Colex means lexicographic on the reversed strings (reading right to left). The illustration supposes that every sequence in the table ends with the same suffix, represented by $*S$.

Nucleotide Sequence	Binary Encoding (LSB … MSB)	Integer Value
CATGCCA$*S$	$10001101101000*S_{\mathrm{bin}}$	$1{,}457 + 2^{14}*S_{\mathrm{bin}}$
ATCGCGA$*S$	$00111001100100*S_{\mathrm{bin}}$	$2{,}460 + 2^{14}*S_{\mathrm{bin}}$
GCTGTGA$*S$	$01101101110100*S_{\mathrm{bin}}$	$2{,}998 + 2^{14}*S_{\mathrm{bin}}$
TACTAGT$*S$	$11001011000111*S_{\mathrm{bin}}$	$14{,}547 + 2^{14}*S_{\mathrm{bin}}$
AACGCGT$*S$	$00001001100111*S_{\mathrm{bin}}$	$14{,}736 + 2^{14}*S_{\mathrm{bin}}$

When constructing the index and bloom filter, using odd numbers of bits from these integer encodings helps fine-tune memory use (see parameters `-l` and `-m` in figure "Query Speed vs Memory Footprint").

2.4 Canonical Orientation

To avoid running separate queries for the forward and reverse-complement orientation of each input sequence, we query for the orientation whose integer encoding is smaller.

Accordingly, in sorting the **Kmers Table**, the sort key for each (forward, rc) kmer pair must be the element of that pair whose integer encoding is smaller.

The only catch is that the recovered *db_kmer* from the **Alleles Table** may be the reverse-complement of *query*. That can be addressed with an additional 64-bit integer comparison of *reverse_complement(query)* to *db_kmer*. This extra integer comparison is much cheaper than the whole second query it replaces, because it doesn't touch RAM.

2.5   Representing the Index with an Array

Conceptually, the index is a map from nucleotide strings – all possible kmer suffixes – to row ranges in the **Kmers Table**. Constraining the domain of the map to suffixes of a specific length – usually up to 16 nucleotides – makes the RAM requirements for representing the index quite acceptable.

As the database contains 2.8 billion kmers, the 16-bp domain of the index map (with 4.3 billion possible suffixes) is densely covered. Therefore, a simple linear array is better-suited to represent this map than, say, a hash table.

Each entry of the **Index Array** is comprised of a 48-bit *range_start* and 16-bit *range_length* representing a range of rows in the **Kmers Table** that all share the same suffix.

To lookup a suffix string *S*, first encode *S* as an integer *I*, then find element *I* in the **Index Array**. If that element is zero, the suffix does not occur in the database. Otherwise, the element's *range_start* and *range_length* tell you which **Kmers Table** rows to examine.
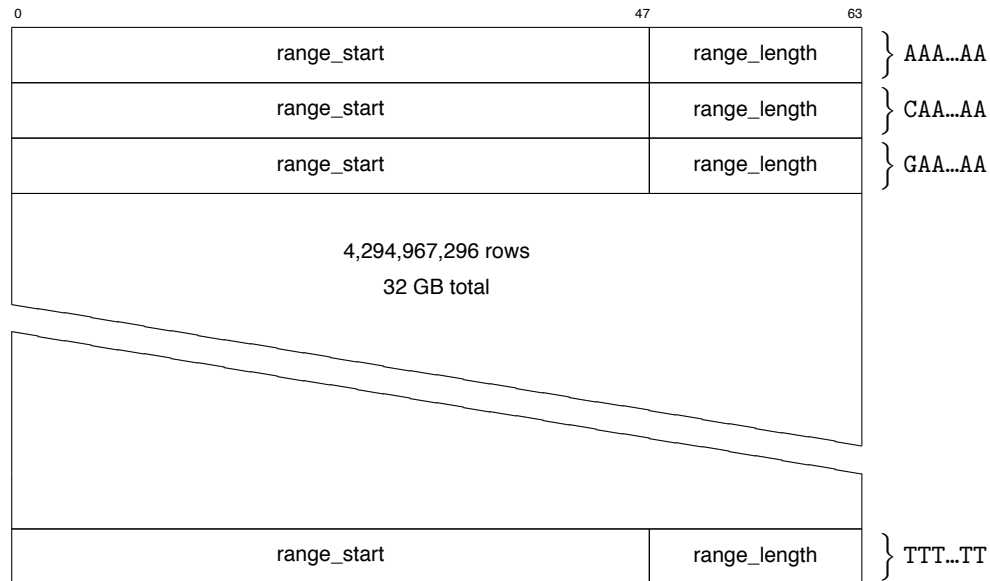
The suffix length and size of the index array are configurable, with defaults that maximize performance for the available RAM.

3 Performance and Scalability Measurements

All measurements reflect full utilization of all cores on the respective machines over most of the execution time. Execution is never I/O bound.

Method	Environment	CPU cores	RAM	Samples	Reads†	Reads / sec
Bowtie+Pileup	Server	24	384 GB	36	173 million	18,036
GT-Pro	Server	24	384 GB	24	72 million	999,158*
GT-Pro	Laptop	8	32 GB	16	24 million	212,203*

† Average read length is 148 base pairs after QC.

* Each GT-Pro measurement is the average of 8 repetitions.

3.1 Baseline Reference

We compared the performance of GT-Pro to a method that produces equivalent results using read alignment and pile-up counting.

The method runs bowtie first, which can align 61,401 reads per second on our 24-core server.

Then it runs pileup with PySAM at 25,528 reads per second utilizing all 24 cores.

This yields 18,036 reads per second, which is 55.4 times slower than GT-Pro on the same machine.

3.2 Sample Selection

Performance was measured on nucleotide sequences from SRA study SRP110665, human gut microbiome fecal samples from Hadza hunter-gatherers in Tanzania, available at `https://www.ncbi.nlm.nih.gov/Traces/study/?acc=SRP110665`.

The selection of samples and reads is arbitrary. For bowtie and pileup, the chosen 36 samples were processed in their entirety. For GT-Pro laptop, each sample contributed just its first 1.5 million reads. For GT-Pro server, the first and the last 1.5 million reads were used.

3.3 Reducing Variance in the Compute Environment

Great care has been taken to obtain reliable and consistent performance measurements from run to run. In the server environment, this involves choosing a machine with a single NUMA node, using the `taskset` utility to ensure each CPU core runs the same number of threads, and disabling the Linux memory defragmenter via

```
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

The effectiveness of these measures is evident in figure "Query Speed vs Memory Footprint", where the 8 distinct measurements for every server configuration overlap closely. These system parameters are not controlled on the MacBook Pro laptop, and the same figure reflects noticeable variance in the laptop data.

The reduced variance in server performance enables quick evaluation of even minute implementation choices, resulting in the simplest possible implementation that does not compromise performance.

The same server environment is used for all measurements, including Bowtie + Pileup, with continuous CPU and I/O monitoring via `sar` and `htop` to assure full machine utlization.

Users of GT-Pro do not need to tweak their Linux settings to achieve peak performance. In fact, a freshly booted Linux system without any tweaks should register even higher performance. However, that performance will tend to degrade and vary over time. Our tweaks ensure it remains consistent for benchmarking.

No tweaks are applied to the laptop environment.

3.4 Multicore Performance

We varied the number of CPU cores while controlling the number of threads per core using the Linux `taskset` utility. Each config ran 8 times.

System performance scaled linearly with the number of cores and was higher with hyperthreading.

Multicore performance, with and without hyperthreading
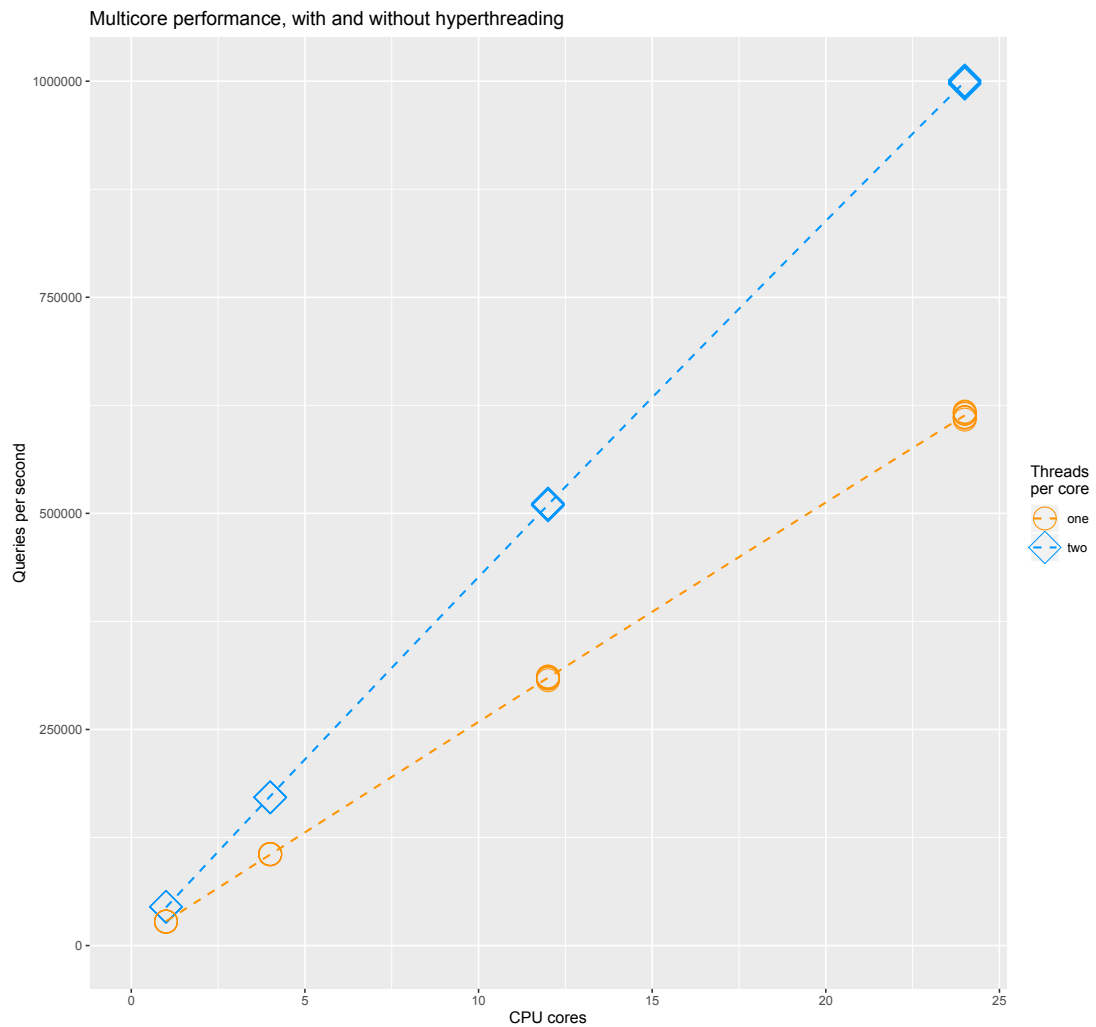
3.5 Multicore Efficiency

We varied the number of CPU cores while controlling the number of threads per core using the Linux `taskset` utility. Each config ran 8 times.

Per-core performance remained relatively constant, and was higher with hyperthreading.
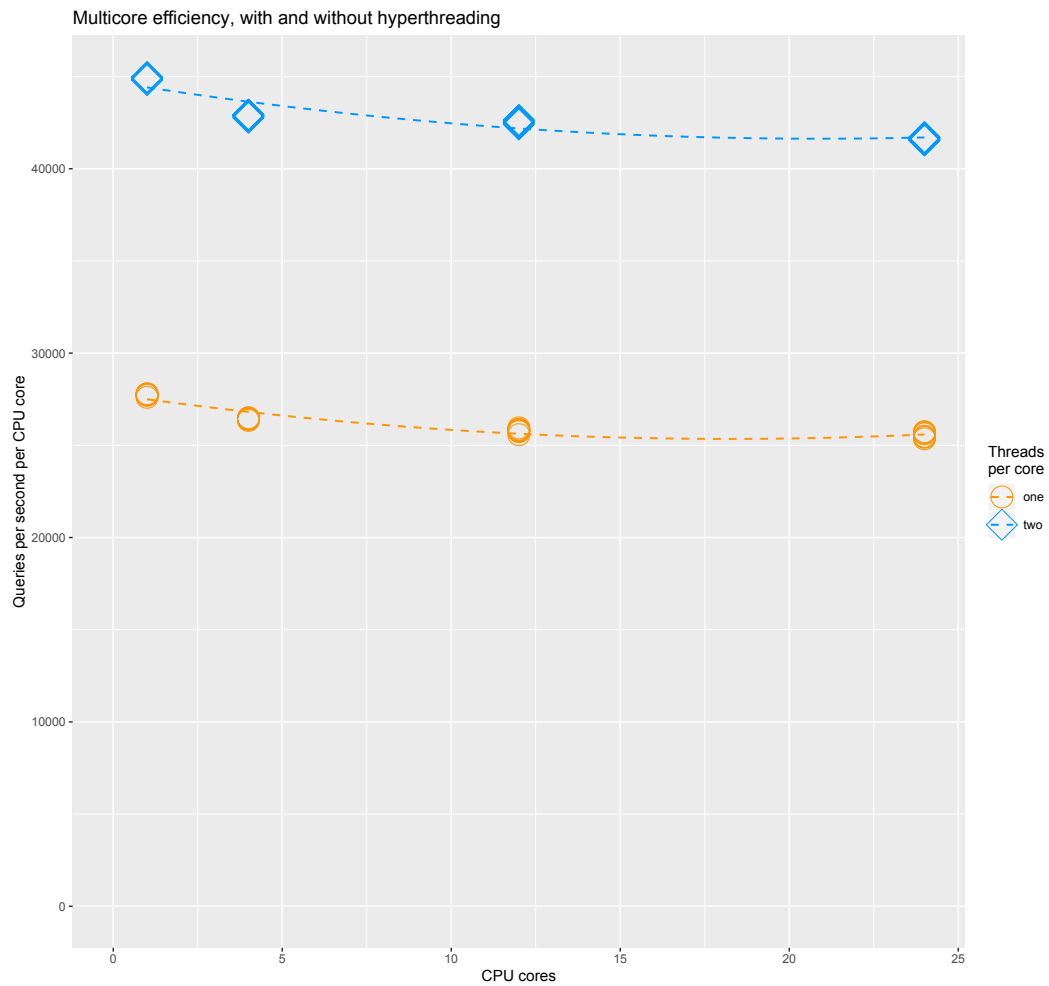
Multicore efficiency, with and without hyperthreading

3.6   Server Parameter Sweeps

We tried many combinations of index and bloom filter sizes (command line parameters -l and -m) to characterize the configuration that yields maximum performance as a function of memory footprint on our server test machine.

The highlighted values for -l and -m win over the greyed-out values by providing the same or better performance with smaller memory footprint.

The 8 measurements averaged to each value in this table are plotted separately on figure "Query Speed vs Memory Use".

l	m	queries per second	queries per second per core	max resident megabytes	user seconds	system seconds	elapsed seconds
32	37	993,159.92	41,381.66	64,746.30	3,322.31	14.59	74.77
32	36	999,157.69	41,631.57	56,555.11	3,300.59	14.27	74.03
32	35	945,716.57	39,404.86	52,460.25	3,490.68	14.25	77.95
32	34	853,896.40	35,579.02	50,418.49	3,859.21	14.13	86.07
31	37	958,976.88	39,957.37	48,354.45	3,438.13	14.08	76.76
30	37	902,582.74	37,607.61	40,166.84	3,646.38	13.80	81.16
31	36	954,573.56	39,773.9	40,164.82	3,448.83	13.71	76.80
31	35	895,411.79	37,308.82	36,076.80	3,676.91	13.64	81.64
29	37	839,179.48	34,965.81	36,069.04	3,878.48	13.64	87.04
31	34	794,859.62	33,119.15	34,042.26	4,152.01	13.50	91.74
30	36	886,687.01	36,945.29	31,979.01	3,699.96	13.54	82.29
30	35	820,213.31	34,175.55	27,887.75	3,999.37	13.29	88.72
29	36	814,371.96	33,932.17	27,882.21	4,005.96	13.48	89.35
30	34	716,264.82	29,844.37	25,839.74	4,596.93	13.27	101.38
29	35	742,123.02	30,921.79	23,788.04	4,395.80	13.21	97.81
29	34	633,866.13	26,411.09	21,739.37	5,175.91	13.15	114.30

3.7 Laptop Parameter Sweeps

We tried many combinations of index and bloom filter sizes (command line parameters -l and -m) to characterize the configuration that yields maximum performance as a function of memory footprint on our laptop test machine.

The highlighted values for -l and -m win over the greyed-out values by providing the same or better performance with smaller memory footprint.

The 8 measurements averaged to each value in this table are plotted separately on figure "Query Speed vs Memory Use".

l	m	queries per second	queries per second per cpu core	max resident megabytes	user seconds	system seconds	elapsed seconds
30	36	26,174.69	3,271.84	28,369.30	1,128.52	1,465.36	929.34
30	35	212,202.90	26,525.36	26,501.61	1,563.03	38.97	115.14
29	36	195,507.65	24,438.46	26,452.43	1,702.25	38.51	124.47
30	34	195,911.25	24,488.91	24,478.62	1,803.03	27.76	124.01
28	36	162,318.44	20,289.80	24,431.18	2,125.50	34.95	149.53
30	33	160,803.56	20,100.45	23,444.82	2,238.51	28.04	150.74
30	32	125,029.91	15,628.74	22,928.46	2,908.98	28.64	193.47
30	31	97,022.85	12,127.86	22,707.12	3,734.34	37.19	248.76
29	35	199,770.81	24,971.35	22,441.26	1,766.38	26.81	121.69
29	34	160,062.28	20,007.79	20,426.87	2,234.41	25.24	151.30
28	35	148,754.99	18,594.37	20,409.00	2,405.93	25.54	162.60
29	33	121,713.55	15,214.19	19,366.31	2,889.60	30.22	198.69
29	32	93,889.46	11,736.18	18,843.75	3,827.70	33.59	257.59
29	31	72,596.65	9,074.58	18,617.81	4,990.75	36.64	332.75
28	34	120,159.31	15,019.91	18,417.15	3,009.71	26.20	201.54
28	33	94,361.57	11,795.20	17,368.82	3,851.05	28.72	255.67
28	32	67,013.60	8,376.70	16,875.52	5,427.62	34.01	359.51
28	31	50,261.59	6,282.70	16,640.05	7,218.18	43.15	478.77

3.8 Query Speed Vs Memory Footprint

The winning `-l` and `-m` parameter configs from our server and laptop parameter
sweeps were plotted to portray the relationship of performance to memory use.

Each config ran 8 times. The server environment has low variance, so most
server measurements overlap. The laptop environment is very noisy, and one
of the `-l 30 -m 35` laptop measurements is an outlier bounded by I/O.

Query Speed vs Memory Use
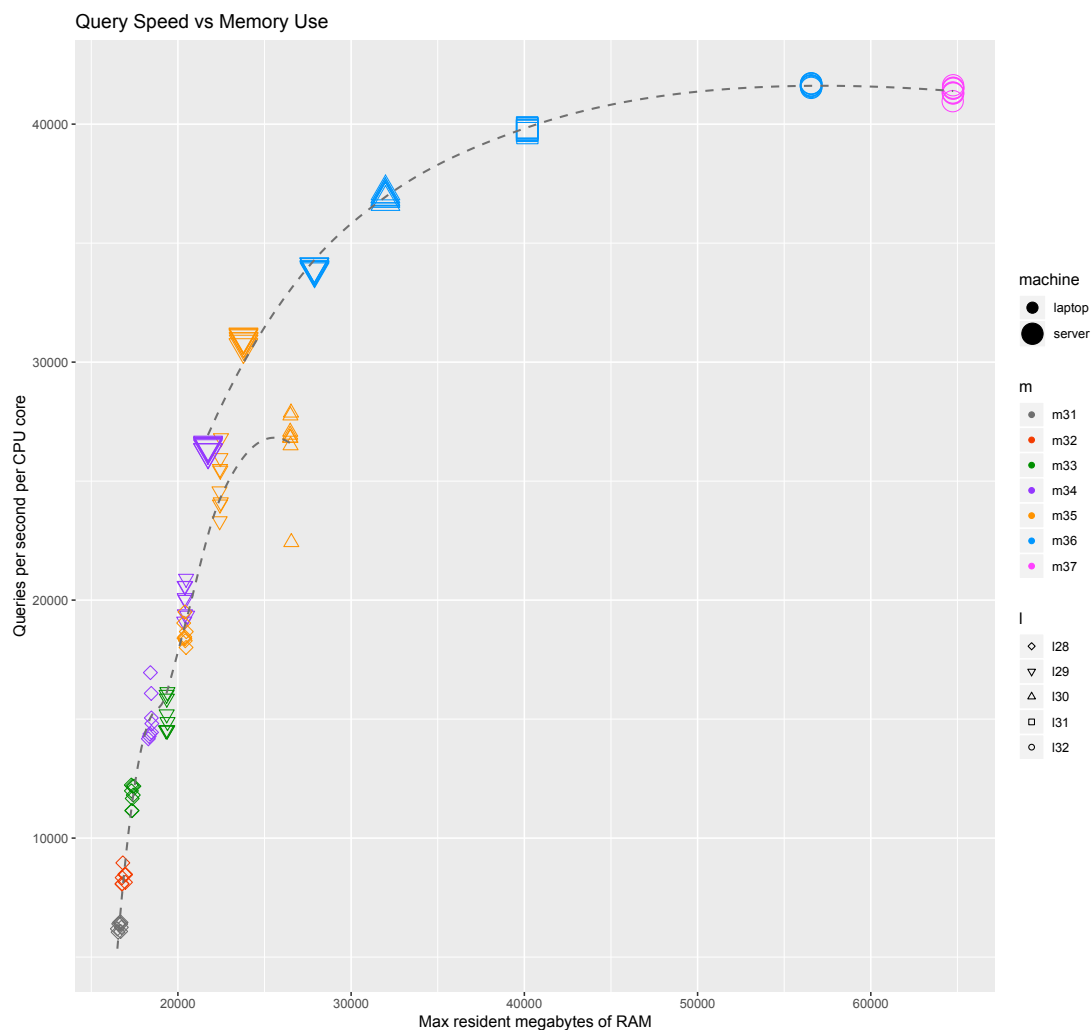
3.9 Server specs

Machine:

- AWS r5.12xlarge.

- 24 physical CPU cores (48 vCPU).

- Intel 8175M CPU @ 2.50GHz.

- Sustained all core Turbo CPU clock speed of up to 3.1 GHz.

- 384 GB RAM

- EBS gp2 RAID array providing 780 MB/sec bandwidth.

Operating system:

- Ubuntu 7.4.0 – 18.04.01.

Compiler:

- G++ 7.4.0.

```
> g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


> lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              48
On-line CPU(s) list: 0-47
Thread(s) per core:  2
Core(s) per socket:  24
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
Stepping:            4
CPU MHz:             1341.926
BogoMIPS:            5000.00
```

```
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            33792K
NUMA node0 CPU(s):   0-47
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good
nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq monitor ssse3 fma cx16
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor
lahf_lm abm 3dnowprefetch invpcid_single pti fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms
invpcid rtm mpx avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl
xsaveopt xsavec xgetbv1 xsaves ida arat pku ospke
```

3.10 Laptop specs

Machine:

- Apple MacBook Pro (15-inch, 2019)
- 8 physical CPU cores (16 vCPU).
- Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz
- Max Turbo CPU clock speed of up to 5.0 GHz.
- 32 GB 2400 MHz DDR4 RAM.
- 2TB APPLE SSD AP2048M

Operating system:

- Mac OS X 10.14.6

Compiler:

- Apple LLVM version 10.0.1 (clang-1001.0.46.4)

```
> g++ --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr
              --with-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 10.0.1 (clang-1001.0.46.4)
Target: x86_64-apple-darwin18.7.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin

> sysctl -n machdep.cpu.brand_string
Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz
```