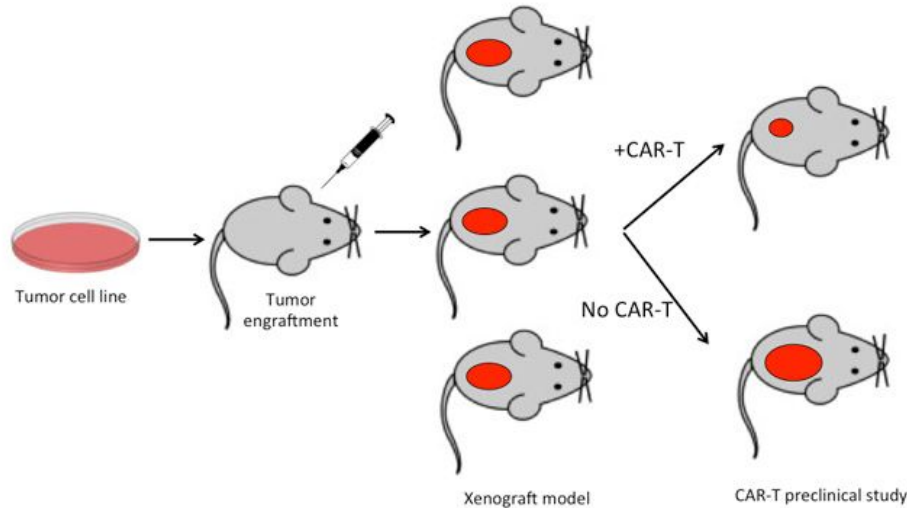# Fast lightweight accurate xenograft sorting

Jens Zentgraf & Sven Rahmann
Genome Informatics, Institute of Human Genetics
University of Duisburg-Essen, Essen, Germany

WABI 2020, 07.-09. September 2020

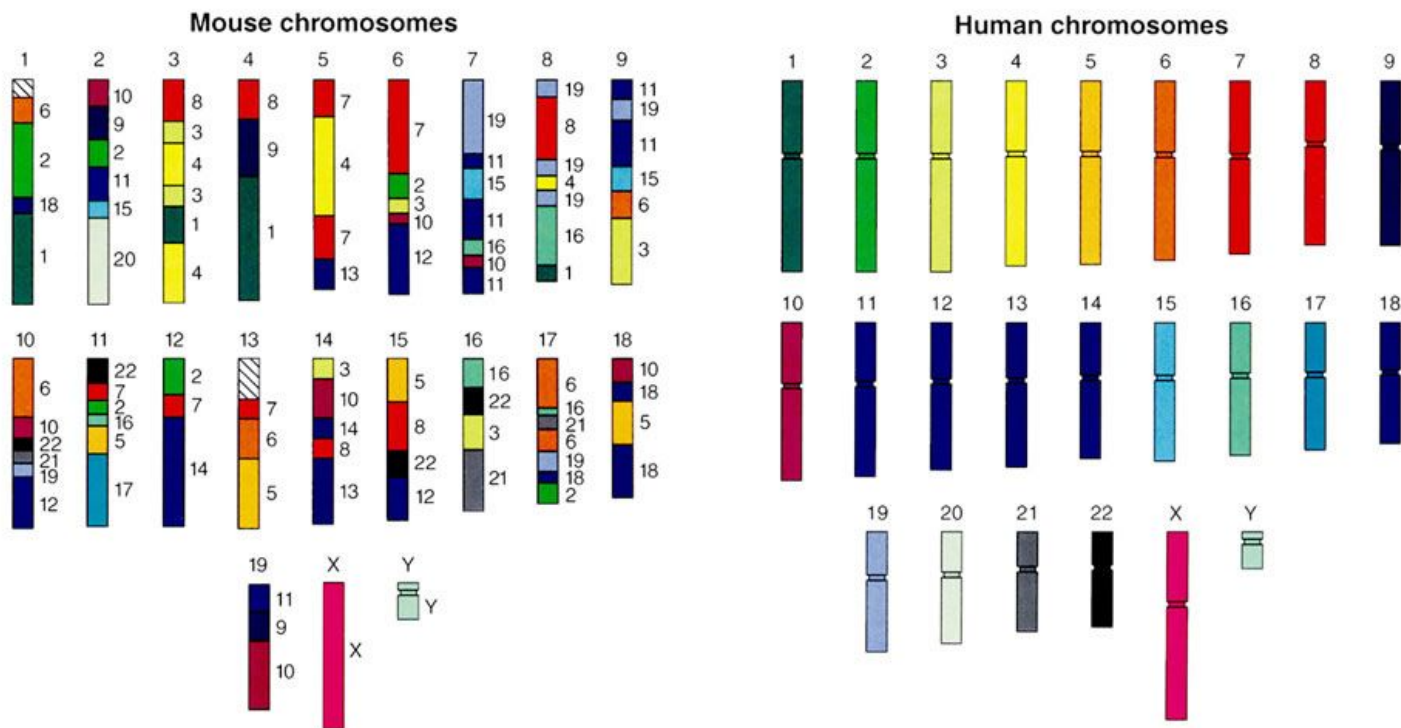# (Patient-derived) xenografts



- tumor cell lines
  or patient tumor samples
  implanted in mice
- study tumor heterogeneity,
  evolution
- sequencing of samples
- mixture of human+mouse DNA
- First task: separate/sort reads
  ("xenograft sorting"), or:
  extract graft (human) reads

# Mouse and Human Genetic Similarities



Source: https://public.ornl.gov/site/gallery/originals/
Mouse_and_Human_Genetic_Similarities_-_original.jpg

Courtesy Lisa Stubbs
Oak Ridge National Laboratory

YGA 98-075R2

# Problem: Human-Aligned Mouse Alleles (HAMAs)

- mouse reads may align to human genome
- may lead to false human (tumor) variant calls
- oncogenes particularly prone to this effect
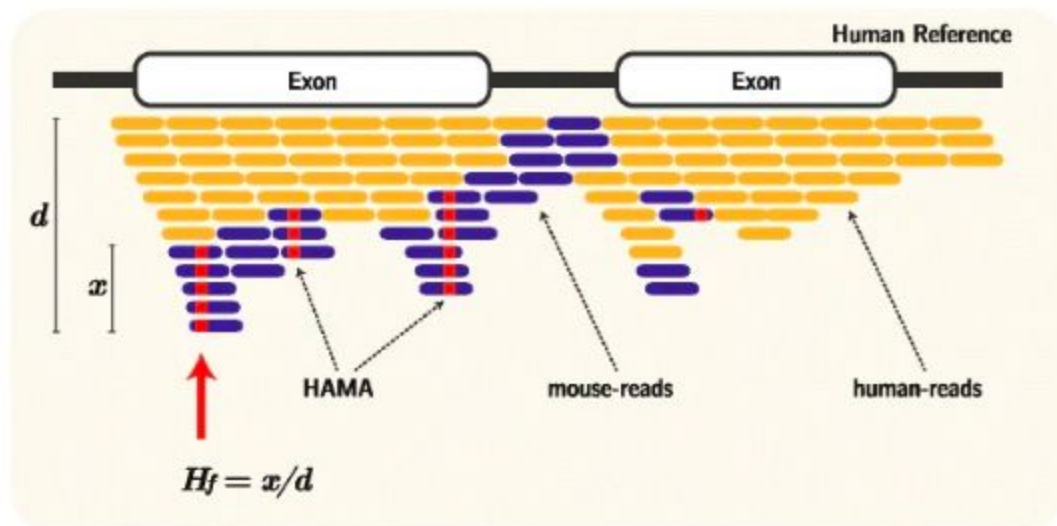
S. Y. Jo, E. Kim, and S. Kim. **Impact of mouse contamination in genomic profiling of patient-derived models and best practice for robust analysis.** *Genome Biology*, 20(1):Article 231, Nov 2019.



Human Reference

Exon    Exon

$d$

$x$

HAMA    mouse-reads    human-reads

$H_f = x/d$

# The xenograft sorting problem

**Given:** sequenced xenograft sample (reads from two species),
paired-end or single-end,
genomic or transcriptomic reads,

sort the reads into five categories according to species of origin:
host (mouse), graft (human), both, neither, ambiguous

or: partially sort using fewer categories (host, graft, other),
or: count how many reads are in each category,
or: filter (select) only graft (human) reads.

Two kinds of tools: aligned BAM vs. raw FASTQ input

| Tool | Input | Operations | Lang. |
|------|-------|-----------|-------|
| *XenofilteR* | aligned BAM | filter | R |
| *Xenosplit* | aligned BAM | filter, count | Python |
| *Bamcmp* | aligned BAM | partial sort | C++ |
| *Disambiguate* | aligned BAM | partial sort | Python or C++ |
| *BBsplit* | raw FASTQ | partial sort | Java |
| *xenome* | raw FASTQ | count, sort | C++ |
| *xengsort* | raw FASTQ | count, sort | Python |

# *k*-mer methods for xenograft sorting

- Partition each read into its *k*-mers

- Look up information on each *k*-mer in a table
  [*k*-mer ↦ human | mouse | both]

- Absent k-mers occur in neither species.

- Aggregate *k*-mer information into a statement
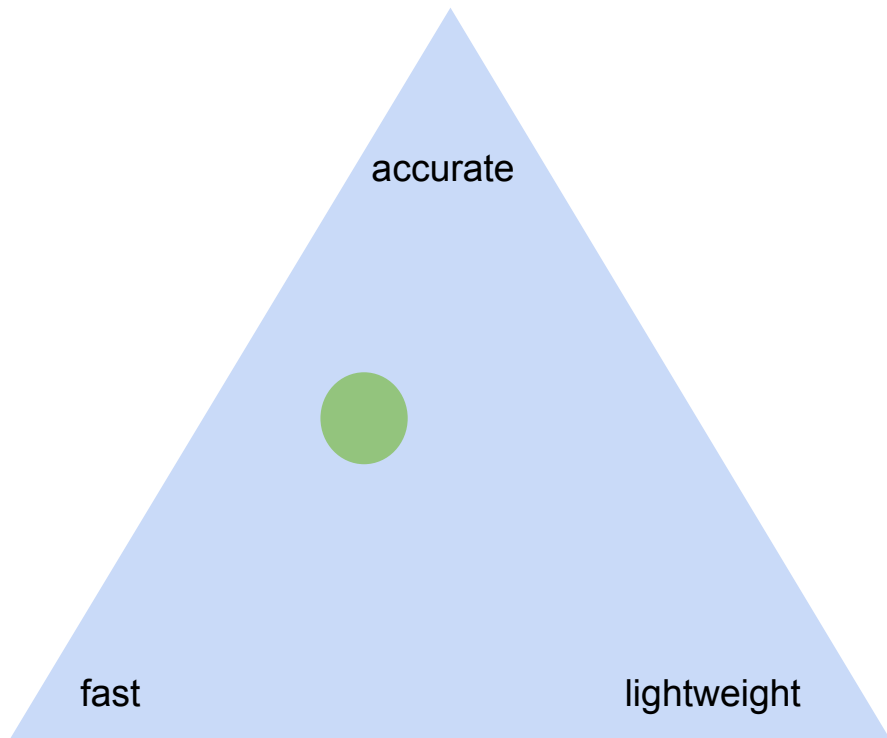  about the read [e.g., majority vote]

GATTCATGC...
GATTC
  ATTCA
    TTCAT
      TCATG
        CATGC
          ......

# Goal: "Fast lightweight accurate xenograft sorting"



**fast:**
- slow random memory accesses
- 3-way bucketed Cuckoo hashing
- buckets fit within a cache line

**lightweight (small memory footprint):**
- 4.5 billion 25-mers + values
- high load (little wasted space)
- quotienting

**accurate:**
- identical + highly similar sequences
- "weak" $k$-mers
- multi-level decision rule

# 3-way Cuckoo hashing with 4-buckets

- 3 hash functions:
- each maps a *k*-mer to a bucket.
- Each bucket can store up to 4 elements.
- Idea: bucket fits within a cache line.

- 12 possible locations for each element.
- At worst 3 memory lookups (cache misses), often only 1 or 2.
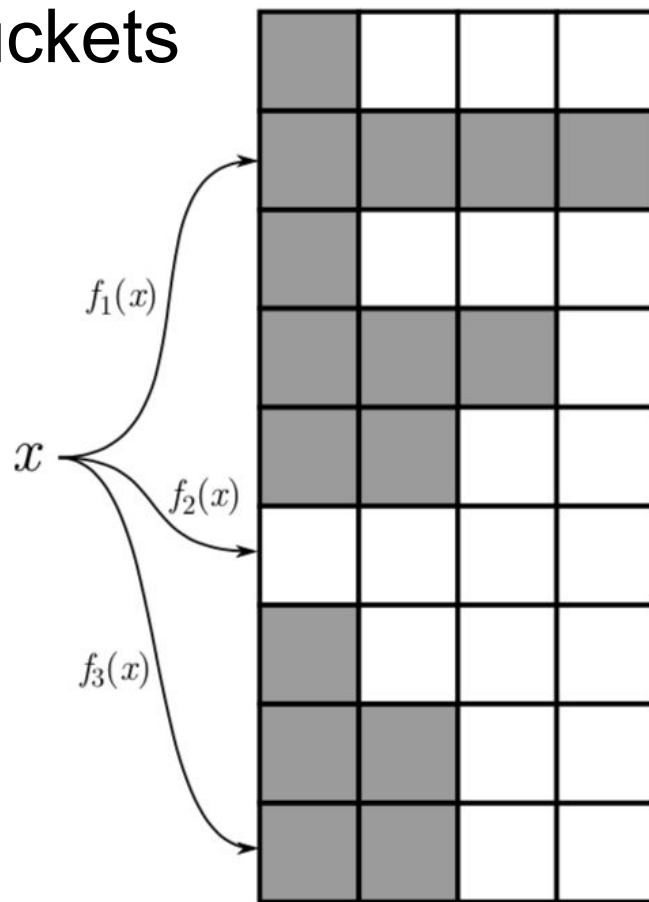
# Insertion by random walk

- Insert $x$: try buckets $f_1(x)$, $f_2(x)$, $f_3(x)$ in order;
  insert into first bucket with space available.

- If all full, evict a random element,
  place current element into now free slot.

- Re-insert evicted element into different slot.

- May cause another eviction…
  ⇒ random walk through table.

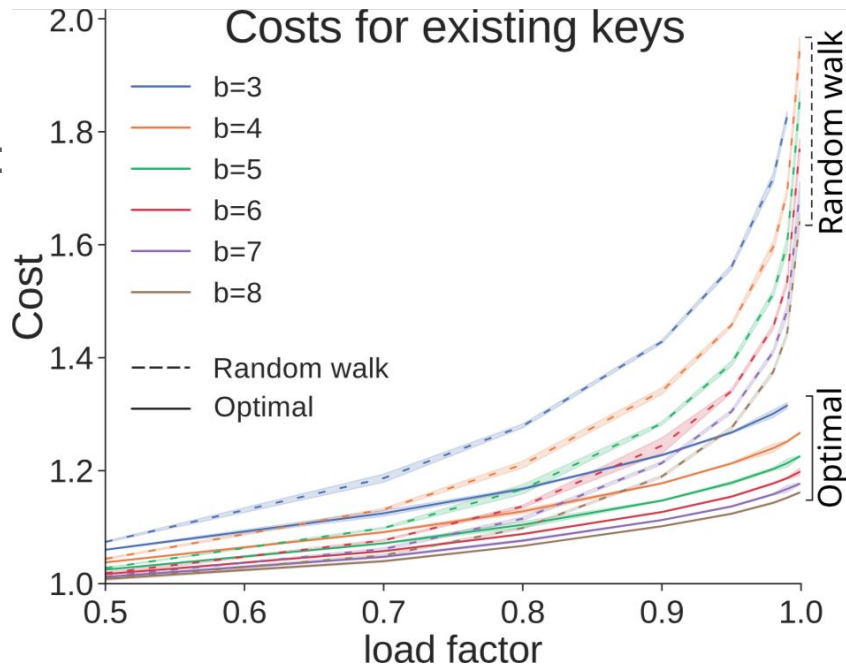- Limit length of walk (e.g. 500 steps).
  Fail if limit reached.
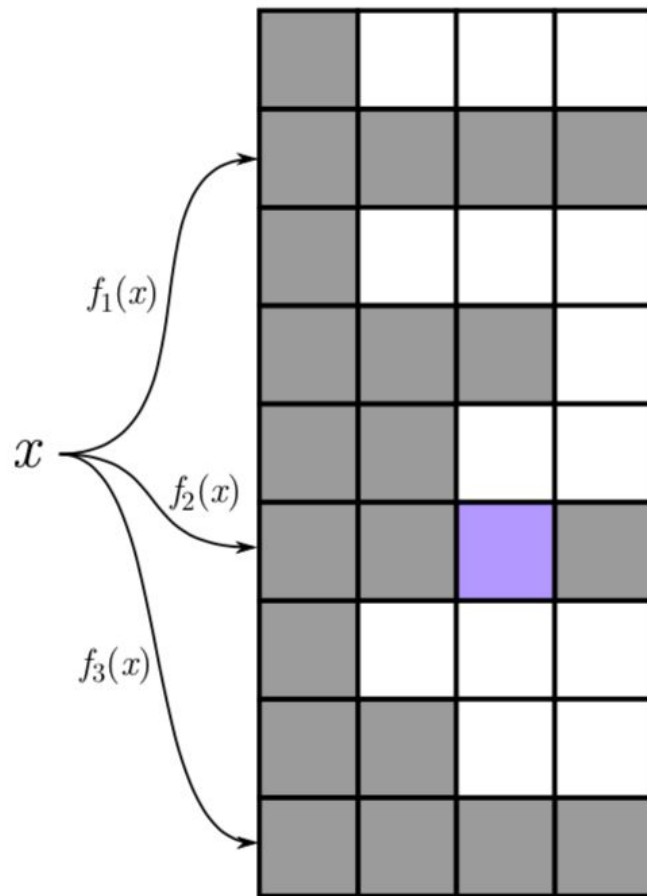
# Speed vs. space: High vs. very high loads

($h$,$b$) = (3,4) allows loads up to 99.9%.

Lower loads offer better choice distribution:
more elements at their first choice;
lower average cost (cache misses).

Placement can be optimized exactly
(Zentgraf et al., ALENEX 2020).

Random walk degrades near 100%.
At 88%, random walk performs ok.

# Weak *k*-mers

Host or graft *k*-mers with a close neighbor (Hamming distance 1)
in the other species are not as reliable (**"weak"**):
A single nucleotide variation suffices to switch species.

After building the hash table, we mark weak *k*-mers.

**Value set of size 5:**  host, weak host, graft, weak graft, both.
Each *k*-mer in the table has exactly one of these values (3 bits).

Fast method to find the Hamming-1 neighbors of each *k*-mer (see paper).

*Xenome*: similar concept with 4 values: host, graft, both, marginal.

# Saving space with quotienting

**Keys:** canonical codes of 25-mers (50 bits)
**Values:** species (5 classes: 3 bits)

4.5 billion k-mers: reference genomes, alternative alleles, cDNA transcripts:
53 bits per entry, load 0.88: **33.88 GB** for hash table 😩

Quotienting to the rescue:

- Do not store full keys (k-mers), but only "quotients" (here 20 bits),
  plus hash function choice (2 bits) plus values (3 bits) → 25 bits per entry:

**15.98 GB** for hash table 😃
(could be slightly reduced by higher load, value compression, etc.)

# Quotienting: Details

Keys are encoded canonical $k$-mers (half of set $[4^k] := \{0, .., 4^k-1\}$).

**Step 1:** Bijective randomizing function $[4^k] \rightarrow [4^k]$ with *a* odd

$$g_{a,b}(x) := [a \cdot (\mathrm{rot}_k(x) \ \mathrm{xor} \ b)] \ \mathrm{mod} \ 4^k$$

**Step 2:** Map to buckets (simply mod $p$: number of buckets). Define

$$f(x) := g_{a,b}(x) \ \mathrm{mod} \ p \quad \text{and} \quad q(x) := g_{a,b}(x) \ // \ p \ .$$

Then $x$ can be uniquely reconstructed
from $f(x)$ ("hash value, "bucket number") and $q(x)$ ("fingerprint", "quotient").
Sufficient to store $q(x)$ in bucket $f(x)$ (and which hash function was chosen).

# Build time [min] & space [GB]

| tool | $k$ | build CPU | build wall | mark CPU | mark wall | total CPU | total wall | mem final | size peak |
|---|---|---|---|---|---|---|---|---|---|
| xengsort | 23 | 50 | 50 | 591 | 176 | 641 | 226 | 12.8 | 17.3 |
| xengsort | 25 | 53 | 53 | 437 | 158 | 490 | 211 | 15.9 | 20.4 |
| xengsort | 27 | 51 | 51 | 495 | 214 | 546 | 265 | 17.3 | 21.8 |
| xenome | 25 | 992 | 151 | 2338 | 356 | 3626 | 552 | 31.2 | 57.1 |
| XenofilteR | – | 528 | 658 | – | – | 528 | 658 | 13.0 | 22.0 |

*xengsort*: 1 thread for build, 8 for mark
*xenome*: 8 (9) threads for build and mark
*XenofilteR*: 8 threads (bwa index)

# Read classification

- Partition read into its *n* valid k-mers

- Look up class of each *k*-mer and count:
    - *h*, *h*':     *k*-mers in read belonging to "host", "weak host"
    - *g*, *g*':     *k*-mers in read belonging to "graft", "weak graft"
    - *b*:           *k*-mers in read belonging to both species
    - *x*:           *k*-mers in read belonging to neither species

# Read classification using ($h$, $h'$, $g$, $g'$, $b$, $x$; $n$)
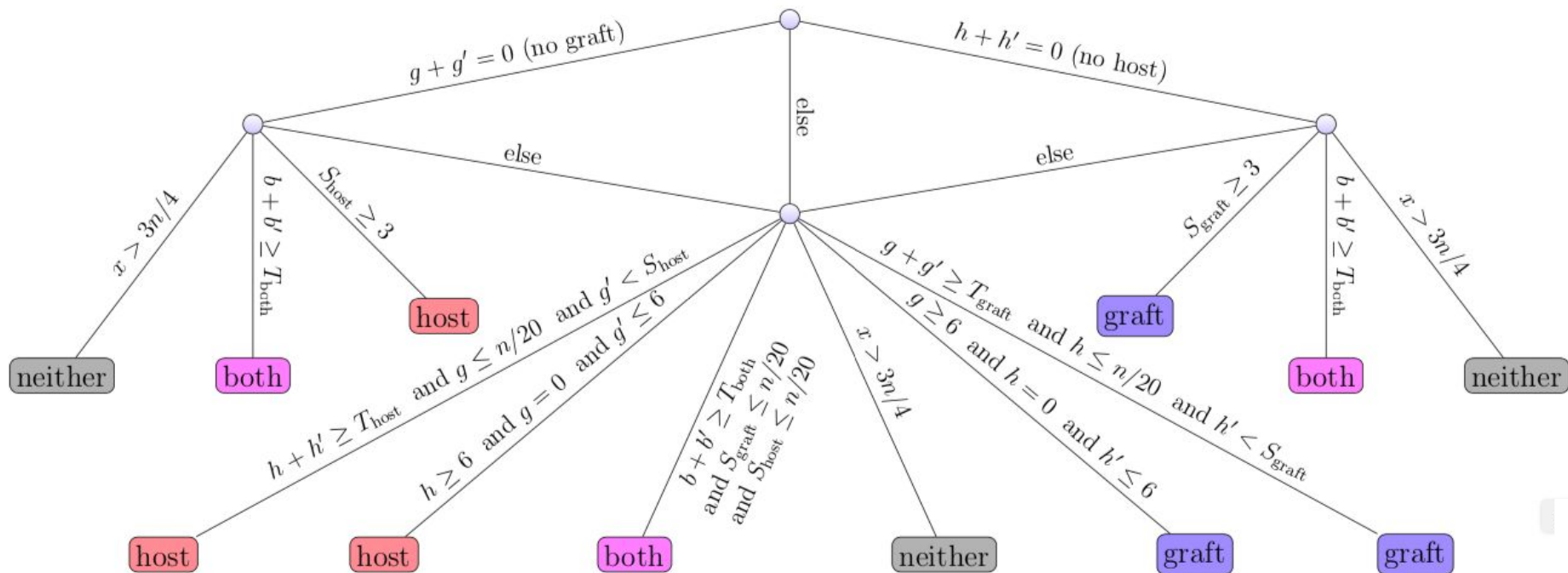
# Quick mode heuristic

(inspired by a similar shortcut in kallisto)

- Examine 3rd and 3rd-last *k*-mer in read and look up classes.
- If classes agree, classify read accordingly.
- Otherwise, count all k-mers and use decision rule tree.

# Results: Comparison of tools

| Tool | Input | Operations | Lang. |
|---|---|---|---|
| *XenofilteR* | aligned BAM | filter | R |
| *Xenosplit* | aligned BAM | filter, count | Python |
| *Bamcmp* | aligned BAM | partial sort | C++ |
| *Disambiguate* | aligned BAM | partial sort | Python or C++ |
| *BBsplit* | raw FASTQ | partial sort | Java |
| *xenome* | raw FASTQ | count, sort | C++ |
| *xengsort* | raw FASTQ | count, sort | Python |

# Human dataset

GIAB human matepair
dataset (Ashkenazim trio;
1258 million read pairs).

Almost all graft (correct).
"Neither" is mostly PhiX.

Quick mode gives almost
identical results.

Xenome sometimes
says "ambiguous".
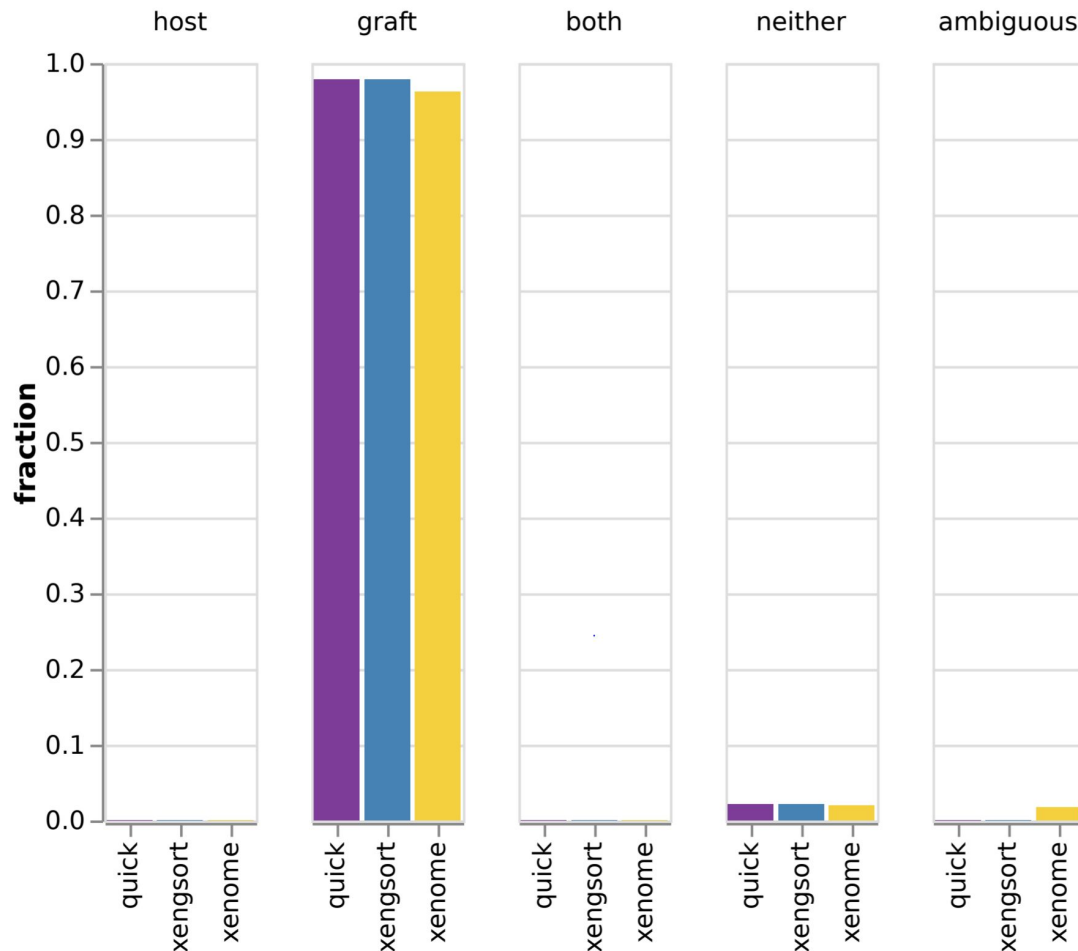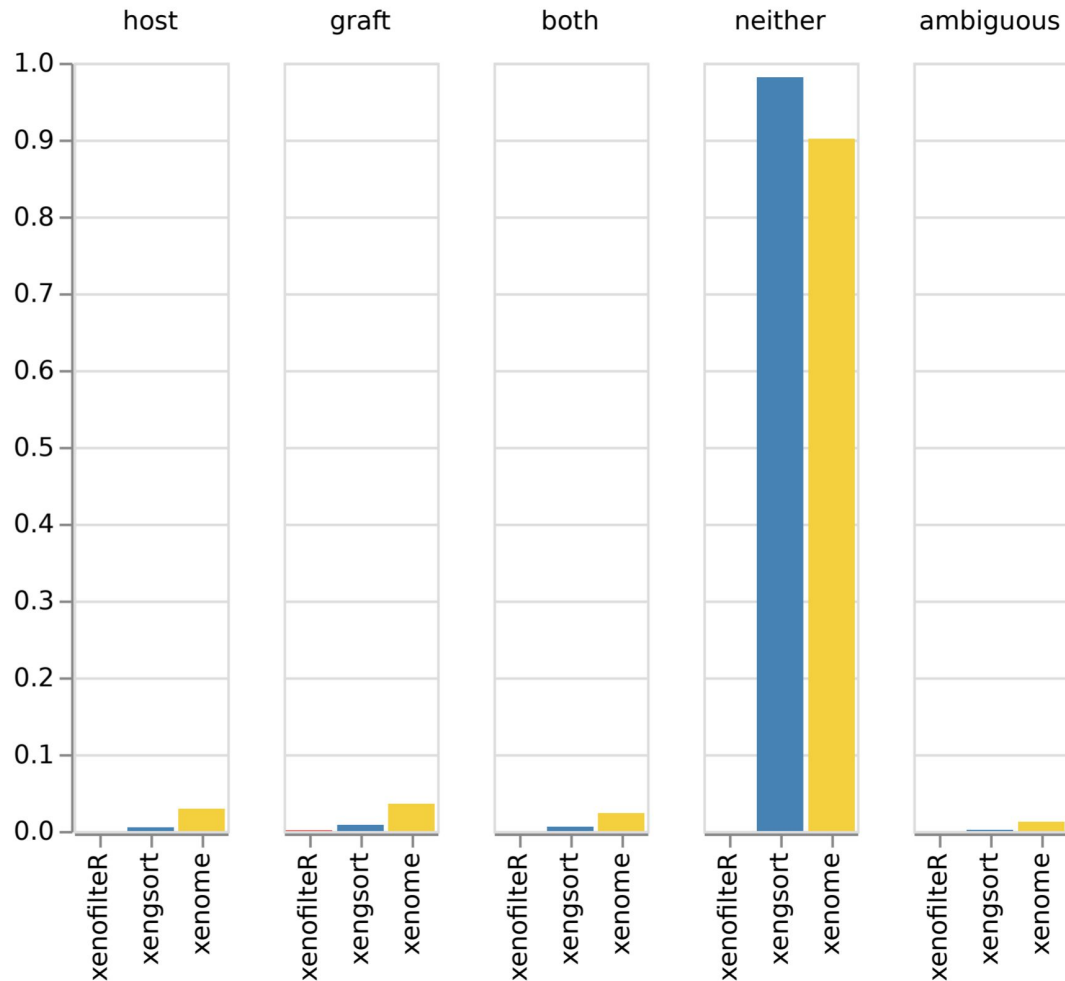
# Chicken dataset

Illumina-sequenced
chicken genome.

XenofilteR only extracts
graft (human) reads,
remainder not classified.
Finds none (correct).
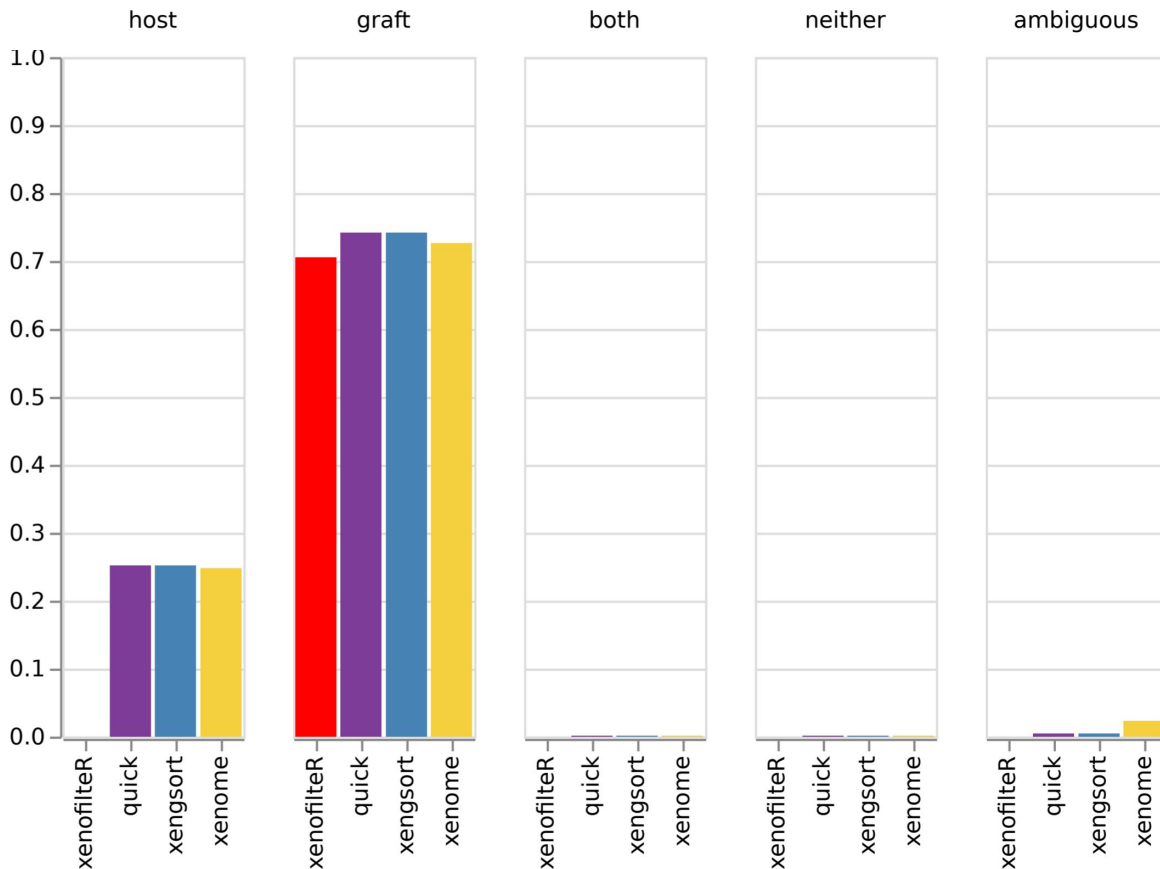
xengsort:
Almost all neither (correct).

xenome: 10% host, graft, both
(lower specificity).

# PDX dataset

174 RNA-seq PDX samples
(human tumor in mouse)
from Jens Siveke,
University Hospital Essen.

XenofilteR only extracts
graft (human) reads,
remainder not classified.

# 174 PDX datasets: Running times [CPU minutes]



**Sum of running times [min]**

| | |
|---|---|
| bwa+sorting | 284191 |
| xenofilteR | 16043 |
| xengsort | 13555 |
| **sort** xenome | 59691 |
| **count** xengsort | 13285 |
| xengsort opt | 11516 |
| xengsort f99 | 14667 |
| xengsort f99 opt | 11824 |
| xengsort quick | 5713 |
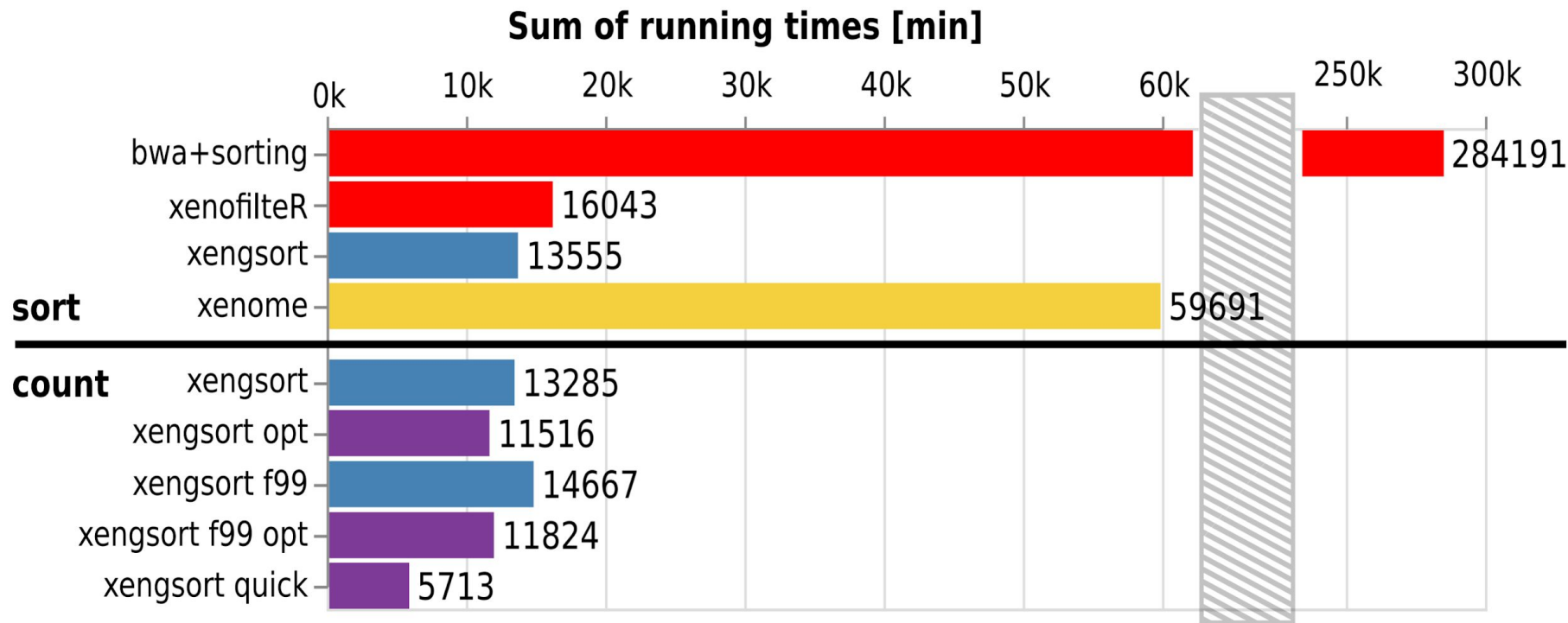
# Summary: Fast lightweight xenograft sorting

- **alignment-free** approach using 25-mers and decision rule
- lightweight on CPU resources, using 3-way bucketed Cuckoo hashing
- Implementation *xengsort* outperforms *xenome*;
  ⅙ of the CPU work, ⅓ of the wall clock time (both 8 threads)
- Typically it takes the same time just to scan the BAM files (*XenofilteR*)
- 25-mer table fits into 16 GB RAM, could be made smaller
  (higher load, compacted values and choice indicators).

- More times [CPU min]
  (see paper for datasets)

| dataset / tool | size | XfR + | bwa + | sort | xenome | xengsort |
|---|---|---|---|---|---|---|
| mouse exomes | 307 M | 310 + | 8291 + | 179 | 1823 | 368 |
| human matepair | 1258 M | N/A + | 222939 + | 940 | 9845 | 2463 |
| chicken genome | 251 M | 76 + | 6976 + | 118 | 1273 | 592 |
| leukemia RNA | 1760 M | 778 + | 22111 + | 521 | 5188 | 1680 |
| PDX RNA | 9742 M | 16043 + | 278329 + | 5862 | 59692 | 13555 |

# Summary: Algorithm Engineering for xenograft sorting

- Hash table
- 3-way bucketed Cuckoo hashing
  (with bucket size 4)
- Keys reduced using quotienting
  (part of key stored in bucket number)
- Interesting trade offs:
  Small buckets = small quotients,
  but lower maximum load,
  and fewer keys at first hash choice.
- Several further engineering opportunities
- Find xengsort at gitlab: https://gitlab.com/genomeinformatics/xengsort/
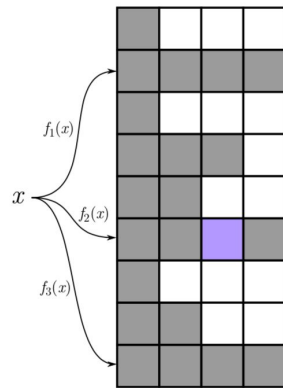


$$g_{a,b}(x) := [a \cdot (\mathrm{rot}_k(x)\ \mathrm{xor}\ b)] \bmod 4^k$$

# Appendix

# Why *k* = 25 ?

| *k*-mers | *k* = 23 | (%) | *k* = 25 | (%) | *k* = 27 | (%) |
|---|---|---|---|---|---|---|
| total | 4 396 323 491 | (100) | 4 496 607 845 | (100) | 4 576 953 994 | (100) |
| host | 1 924 087 512 | (43.8) | 2 050 845 757 | (45.6) | 2 105 520 461 | (46.0) |
| graft | 2 173 923 063 | (49.4) | 2 323 880 612 | (51.7) | 2 395 147 724 | (52.3) |
| both | 18 701 862 | (0.4) | 12 579 160 | (0.3) | 9 627 252 | (0.2) |
| wk host | 132 469 231 | (3.0) | 52 063 110 | (1.2) | 32 445 717 | (0.7) |
| wk graft | 147 141 823 | (3.4) | 57 239 206 | (1.3) | 34 212 840 | (0.7) |

# Why (*h*,*b*) = (3,4) ?

More hash functions (*h*), larger buckets (*b*) have ⊕ and ⊖ effects:

⊕ higher load limit
   [only 50% for standard (2,1)]
   [over 99.9% for (3,4),
   less w/ random walk]

⊖ more worst case cache misses (*h*)

⊖ more search effort per bucket (*b*)

   ■ (3,4) is a good compromise;
   maybe also (2,8).

| *b* \| *h* | 2 | 3 |
|---|---|---|
| 1 | 0.5 | 0.9179352767 |
| 2 | 0.8970118682 | 0.9882014140 |
| 3 | 0.9591542686 | 0.9972857393 |
| 4 | 0.9803697743 | 0.9992531564 |

S. Walzer. Load thresholds for cuckoo
hashing with overlapping blocks.
ICALP 2018, LIPIcs 107:102.