Data Science at Scale:
Scaling Up by Scaling Down and Out (to Disk)

Prashant Pandey
ppandey@berkeley.edu
Berkeley Lab/UC Berkeley
Sequence Read Archive (SRA) database growth

SRA contains a lot of diversity information
Goal: perform sequence searches on the database
Scalability is the bottleneck for data science

Data science applications only looking at a *small portion* of data
Scalable data systems → Scalable data science

Current index size (few TBs)

My goal as a researcher is to build scalable data systems to accelerate and scale data science applications
Three approaches to handle massive data
Three approaches to handle massive data

Shrink it

**Goal:** make data smaller to fit in RAM

**Techniques:**
- Compact & succinct data structures
- Filters, e.g., Bloom, quotient, etc.
Three approaches to handle massive data

**Shrink it**

**Goal**: make data smaller to fit in RAM

**Techniques**:
- Compact & succinct data structures
- Filters, e.g., Bloom, quotient, etc.

**Organize it**

**Goal**: organize data in a disk-friendly way

**Techniques**:
- B-tree
- $B^\varepsilon$-tree
- LSM-tree
Three approaches to handle massive data

**Shrink it**
- **Goal**: make data smaller to fit in RAM
- **Techniques**:
  - Compact & succinct data structures
  - Filters, e.g., Bloom, quotient, etc.

**Organize it**
- **Goal**: organize data in a disk-friendly way
- **Techniques**:
  - B-tree
  - $\text{B}^{\epsilon}$-tree
  - LSM-tree

**Distribute it**
- **Goal**: partition and distribute data on multiple nodes
- **Techniques**:
  - Distributed hash table
  - Distributed key-value store
Data structures & Algorithms

- **(Counting) Quotient Filter**
  - SIGMOD ‘17, arXiv ‘17
- **Buffered Count-Min Sketch**
  - ESA ‘18
- **Order Min Hash**
  - ISMB ‘19
Research output

Data structures & Algorithms

(COUNTING) Quotient Filter
SIGMOD ‘17, arXiv ‘17

Buffered Count-Min Sketch
ESA ‘18

Order Min Hash
ISMB ‘19

Bε-trFS file system
FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19
Research output

Data structures & Algorithms

- (Counting) Quotient Filter
  - SIGMOD ‘17, arXiv ‘17

- Buffered Count-Min Sketch
  - ESA ‘18

- Order Min Hash
  - ISMB ‘19

Computational biology
- Squeakr, deBGR, Mantis, Rainbowfish, MST-Mantis
  - ISMB ‘17, WABI ‘17, BIOINFORMATICS ‘17, RECOMB ‘18, Cell Systems ‘18, RECOMB ‘19, JCB ‘20

- LSM-Mantis, VaraintStore
  - bioRxiv ‘20, bioRxiv ‘21

Distributed k-mer counting
- IPDPS ‘21

File systems
- BεtrFS file system
  - FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

Bε-tree
Research output

Data structures & Algorithms

(Counting) Quotient Filter
SIGMOD ‘17, arXiv ‘17

Buffered Count-Min Sketch
ESA ‘18

Order Min Hash
ISMB ‘19

Computational biology
Squeakr, deBGR, Mantis, Rainbowfish, MST-Mantis
ISMB ‘17, WABI ‘17, BIOINFORMATICS ‘17, RECOMB ‘18, Cell Systems ‘18, RECOMB ‘19, JCB ‘20

Stream processing
LERTs
arXiv ‘19, SIGMOD ‘20

File systems
BεtrFS file system
FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

Distributed k-mer counting
IPDPS ‘21
In this talk

Data structures & Algorithms

- (Counting) Quotient Filter
  - SIGMOD ‘17, arXiv ‘17

- Buffered Count-Min Sketch
  - ESA ‘18

- Order Min Hash
  - ISMB ‘19

Shrink it

Computational biology
- Squeakr, deBGR, Mantis, Rainbowfish, MST-Mantis
  - ISMB ‘17, WABI ‘17, BIOINFORMATICS ‘17, RECOMB ‘18, Cell Systems ‘18, RECOMB ‘19, JCB ‘20

- LSM-Mantis, VaraintStore
  - bioRxiv ‘20, bioRxiv ‘21

Distributed $k$-mer counting
- IPDPS ‘21

Stream processing
- LERTs
  - arXiv ‘19, SIGMOD ‘20

File systems
- BεtrFS file system
  - FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

13
In this talk

Data structures & Algorithms

Shrink it

- Quotient Filter
  - SIGMOD ‘17, arXiv ‘17

Buffered Count-Min Sketch
- ESA ‘18

Order Min Hash
- ISMB ‘19

Organize it

LERTs
- arXiv ‘19, SIGMOD ‘20

File systems

BεtrFS file system
- FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

Stream processing

Computational biology

Distributed k-mer counting
- IPDPS ‘21

- Squeakr, deBGR, Mantis, Rainbowfish, MST-Mantis
  - ISMB ‘17, WABI ‘17, BIOINFORMATICS ‘17, RECOMB ‘18, Cell Systems ‘18, RECOMB ‘19, JCB ‘20

- LSM-Mantis, VaraintStore
  - bioRxiv ‘20, bioRxiv ‘21

- Distributed k-mer counting
  - IPDPS ‘21

- (Counting) Quotient Filter
  - SIGMOD ‘17, arXiv ‘17

- Buffered Count-Min Sketch
  - ESA ‘18

- Order Min Hash
  - ISMB ‘19

- LERTs
  - arXiv ‘19, SIGMOD ‘20

- BεtrFS file system
  - FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

- Distributed k-mer counting
  - IPDPS ‘21

- (Counting) Quotient Filter
  - SIGMOD ‘17, arXiv ‘17

- Buffered Count-Min Sketch
  - ESA ‘18

- Order Min Hash
  - ISMB ‘19

- LERTs
  - arXiv ‘19, SIGMOD ‘20

- BεtrFS file system
  - FAST ‘15, TOS 15, FAST ‘16, TOS 16, SPAA ‘19

- Distributed k-mer counting
  - IPDPS ‘21
A dictionary maintains a set $S$ from universe $U$.

A dictionary supports membership queries on $S$.

- membership($a$): ✔
- membership($b$): ✗
- membership($c$): ✔
- membership($d$): ✗
A filter is an *approximate* dictionary.

A filter supports *approximate* membership queries on $S$.

- membership($a$): ✓
- membership($b$): ✗
- membership($c$): ✓
- membership($d$): ✓ ✗ false positive

$S$
A filter guarantees a false-positive rate $\varepsilon$

if $q \in S$, return $\checkmark$ with probability $1$

if $q \notin S$, return

\[
\begin{cases}
\times & \text{with probability } 1 - \varepsilon \\
\checkmark & \text{with probability } \leq \varepsilon
\end{cases}
\]

true positive
true negative
false positive

one-sided errors
False-positive rate enables filters to be compact

\[ \text{space} \geq n \log(1/\epsilon) \]

\[ \text{space} = \Omega(n \log |U|) \]
False-positive rate enables filters to be compact.

For most practical purposes: \( \varepsilon = 2\% \), Bloom filter requires \( \approx 8 \text{ bits/item} \).
Bloom filter: a bit array + $k$ hash functions
Bloom filter: a bit array + $k$ hash functions (here $k = 2$)

$h_1(a) = 1$
$h_2(a) = 3$

$h_1(c) = 5$
$h_2(c) = 3$
Bloom filter: a bit array + $k$ hash functions (here $k=2$)

$h_1(b) = 2$
$h_2(b) = 5$
Classic filter: The Bloom filter [Bloom ‘70]

Bloom filter: a bit array + $k$ hash functions (here $k=2$)

- $h_1(d) = 1$
- $h_2(d) = 3$

False positive
Bloom filter are ubiquitous (> 4300 citations)

- Streaming applications
- Databases (MySQL, Oracle, SQL Server)
- Computational biology
- Networking
- Storage systems
Bloom filters have suboptimal asymptotics

<table>
<thead>
<tr>
<th></th>
<th>Bloom filter</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$\approx 1.44 \ n \ \log(1/\epsilon)$</td>
<td>$\approx n \ \log(1/\epsilon) + \Omega(n)$</td>
</tr>
<tr>
<td><strong>CPU cost</strong></td>
<td>$\Omega(1/\epsilon)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>Data locality</strong></td>
<td>$\Omega(1/\epsilon)$ probes</td>
<td>$O(1)$ probes</td>
</tr>
</tbody>
</table>
## Limitations

<table>
<thead>
<tr>
<th>Limitations</th>
<th>Workarounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>No deletes</td>
<td>Rebuild</td>
</tr>
<tr>
<td>No resizes</td>
<td>Guess $N$, and rebuild if wrong</td>
</tr>
<tr>
<td>No filter merging or enumeration</td>
<td>???</td>
</tr>
<tr>
<td>No values associated with keys</td>
<td>Combine with another data structure</td>
</tr>
</tbody>
</table>

Application often work around Bloom filter limitations.

Bloom filter limitations increase system complexity, waste space, and slow down application performance.
• **Store fingerprints compactly in a hash table.**
  ○ Take a fingerprint $h(x)$ for each element $x$.

• **Only source of false positives:**
  ○ Two distinct elements $x$ and $y$, where $h(x) = h(y)$
  ○ If $x$ is stored and $y$ isn’t, $\text{query}(y)$ gives a false positives

\[
\Pr[ x \text{ and } y \text{ collide}] = \frac{1}{2^p}
\]
Storing fingerprints compactly

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$
Storing fingerprints compactly

- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?
Storing fingerprints compactly

- \( b(x) = \) location in the hash table
- \( t(x) = \) tag stored in the hash table

Collisions in the hash table?
- Linear probing
- Robin Hood hashing
Storing fingerprints compactly

- \( b(x) = \) location in the hash table
- \( t(x) = \) tag stored in the hash table

Collisions in the hash table?
- Linear probing
- Robin Hood hashing

\[ h(x) \]  

\[ b(x) \quad t(x) \]  

Bucket index  

Tag

\( b(y) \)  

\( t(y) \)

\( b(y) \) and \( t(y) \) belong to slots 4 or 5?
QF uses two metadata bits to resolve collisions and identify home bucket

The metadata bits group tags by their home bucket
Resolving collisions in the QF [Bender ‘12, Pandey ‘17]

- QF uses two metadata bits to resolve collisions and identify home bucket

- The metadata bits group tags by their home bucket
Resolving collisions in the QF \cite{Bender '12, Pandey '17}

- QF uses two metadata bits to resolve collisions and identify home bucket

The metadata bits enable us to identify the slots holding the contents of each bucket.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t(u)$</td>
<td>$t(v)$</td>
<td>$t(v)$</td>
<td>$t(w)$</td>
<td>$t(x)$</td>
<td>$t(y)$</td>
</tr>
</tbody>
</table>

insert $v$
Quotienting enables many features in the QF

- Good cache locality
- Efficient scaling out-of-RAM
- Deletions
- Enumerability/Mergeability
- Resizing
- Maintains count estimates
- Uses variable-sized encoding for counts [Counting quotient filter]
  - Asymptotically optimal space: $O(\sum |C(x)|)$
Quotient filters use less space than Bloom filters for all practical configurations.

<table>
<thead>
<tr>
<th></th>
<th>Quotient filter</th>
<th>Bloom filter</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$\approx n \log(1/\epsilon) + 2.125n$</td>
<td>$\approx 1.44 n \log(1/\epsilon)$</td>
<td>$\approx n \log(1/\epsilon) + \Omega(n)$</td>
</tr>
<tr>
<td>CPU cost</td>
<td>$O(1)$ expected</td>
<td>$\Omega(1/\epsilon)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Data locality</td>
<td>1 probe + scan</td>
<td>$\Omega(1/\epsilon)$ probes</td>
<td>$O(1)$ probes</td>
</tr>
</tbody>
</table>

The quotient filter has theoretical advantages over the Bloom filter.
Bloom filter: $\sim 1.44 \log(1/\varepsilon)$ bits/element.

Quotient filter: $\sim 2.125 + \log(1/\varepsilon)$ bits/element.

Quotient filters use less space than Bloom filters for all practical configurations.

False-positive rate < 1/64 (or 0.15).
Quotient filters perform better (or similar) to other non-counting filters

- Insert performance is similar to the state-of-the-art non-counting filters
- Query performance is significantly fast at low load-factors and slightly slower at higher load-factors
Quotient filter’s impact in computer science

**Theoretically well-founded** data structures can have a **big impact** on multiple subfields across **academia and industry**

### Computational biology
1. Squeakr
2. deBGR
3. Mantis
4. SPAdes assembler
5. Khmer software
6. MQF
7. VariantStore

### Databases/Systems
1. Counting on GPUs
2. Concurrent filters
3. Anomaly detection
4. BetrFS file system

### Industry
1. VMware
2. Nutanix
3. Apocrypha
4. Hyrise
5. **A data security startup**
Learned “Shrink it”. Now “Organize it”
Open problem in stream processing

- A **high-speed stream** of key-value pairs arriving over time
- **Goal:** report every key **as soon as** it appears $T$ times without missing any
- Firehose benchmark (Sandia National Lab) simulates the stream [https://firehose.sandia.gov/](https://firehose.sandia.gov/)
Why should we care about this problem

*Defense systems for cyber security* monitor high-speed streams for malicious traffic

Malicious traffic forms a small portion of the stream

Automated systems take defensive actions for every reported event

- Catch all malicious events
- Small reporting threshold
- Minimize false positives
Timely event detection problem

- Stream of elements arrive over time
Timely event detection problem

- Stream of elements arrive over time
- An **event** occurs at time $t$ if $S_t$ occurs exactly $T$ times in $(s_1, s_2, \ldots, s_t)$

![Diagram showing events over time](image-url)
Timely event detection problem

- Stream of elements arrive over time
- An *event* occurs at time $t$ if $S_t$ occurs exactly $T$ times in $(s_1, s_2, \ldots, s_t)$

Suppose $T = 4$
Timely event detection problem

- Stream of elements arrive over time
- An event occurs at time $t$ if $S_t$ occurs exactly $T$ times in $(s_1, s_2, \ldots, s_t)$
- In timely event-detection problem (TED), we want to report all events shortly after they occur.

Suppose $T = 4$
Features we need in the solution

- Stream is large (e.g., terabytes) and high-speed (millions/sec)

High throughput ingestion
Features we need in the solution

- Stream is large (e.g., terabytes) and high-speed (millions/sec)
- Events are high-consequence real-life events
- High throughput ingestion
- No false-negatives; few false-positives
- Timely reporting (real-time)
Features we need in the solution

- Stream is large (e.g., terabytes) and high-speed (millions/sec)
- Events are high-consequence real-life events
- Malicious traffic forms a small portion of the stream

High throughput ingestion

No false-negatives; few false-positives

Timely reporting (real-time)

Very small reporting thresholds

Sampling

Danger
One-pass streaming has errors

- **Heavy hitter problem**: report items whose frequency $\geq \varphi N$
- Exact one-pass solution solution requires $\Omega(N)$ space
One-pass streaming has errors

- **Approximate solution**: report all items with count $\geq \phi N$, none with $< (\phi - \varepsilon)N$ [Alon et al. 96, Berinde et al. 10, Bhattacharyya et al. 16, Bose et al. 03, Braverman et al. 16, Charikar et al. 02, Cormode et al. 05, Demaine et al. 02, Dimitropoulos et al. 08, Larsen et al. 16, Manku et al. 02.]

- **Approximate solutions requires**: $\Omega(1/\varepsilon)$

![](diagram.png)

Real time with false-positives!
One-pass streaming has errors

- **Approximate solution**: report all items with count \( \geq \varphi N \), none with \( < (\varphi - \varepsilon)N \) [Alon et al. 96, Berinde et al. 10, Bhattacharyya et al. 16, Bose et al. 03, Braverman et al. 16, Charikar et al. 02, Cormode et al. 05, Demaine et al. 02, Dimitropoulos et al. 08, Larsen et al. 16, Manku et al. 02.]

- **Approximate solutions requires**: \( \Omega(1/\varepsilon) \)

  For Sandia, \( \varphi N \) is a small constant (e.g., 24), So \( \Omega(1/\varepsilon) \) is very very large!!

  Can’t solve in RAM for very small \( \varphi \)

Real time with false-positives!
One-pass solution has:

- Stream is large (e.g., terabytes) and high-speed (millions/sec)
- Events are high-consequence real-life events
- Malicious traffic forms a small portion of the stream

- High throughput ingestion
- No false-negatives; few false-positives
- Timely reporting (real-time)

Very small reporting thresholds
Two-pass streaming isn’t real-time

- A second pass over the stream can get rid of errors
- Store the stream on SSD and access it later

Scales to very small φ but offline!
Two-pass solution has:

- Stream is large (e.g., terabytes) and high-speed (millions/sec)
- Events are high-consequence real-life events
- Malicious traffic forms a small portion of the stream

High throughput ingestion

No false-negatives; few false-positives

Timely reporting (real-time)

Very small reporting thresholds
If data is stored: why not access it?

Why wait for second pass?
Idea: combine Streaming and EM

Use an efficient external-memory counting data structure to scale Misra-Gries algorithm to SSDs
How computations work:
- Data is transferred in blocks between RAM and disk.
- The number of block transfers dominate the running time.

Goal: Minimize number of block transfers
- Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$. 

External memory model [Aggarwal+Vitter ‘08]
Cascade filter: write-optimized quotient filter
[Bender et al. ‘12, Pandey et al. ‘17]

- The Cascade filter efficiently scales out-of-RAM
- It accelerates insertions at some cost to queries
## Cascade filter operations

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
<tr>
<td>Insert</td>
<td>Query</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
<td></td>
</tr>
</tbody>
</table>

< 1 I/O per observation
# Cascade filter operations

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

- **Insert**: $< 1$ I/O per observation
- **Query**: $> 1$ I/O per observation
Cascade filter doesn’t have real-time reporting

But every insert is also a query in real-time reporting!

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O\left(\frac{1}{B}\log\frac{N}{M}\right)$</td>
<td>$O\left(\log\frac{N}{M}\right)$</td>
</tr>
</tbody>
</table>

- **< 1 I/O per observation** 🎉
- **> 1 I/O per observation** 😞
Cascade filter doesn’t have real-time reporting

But every insert is also a query in real-time reporting!

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

Traditional cascade filter doesn’t solve the problem!
We define the time stretch of a report to be

\[ \text{Time stretch} = 1 + \alpha = 1 + \frac{\text{Delay}}{\text{Lifetime}} \]
We define the time stretch of a report to be

\[ \text{Time stretch} = 1 + \alpha = 1 + \frac{\text{Delay}}{\text{Lifetime}} \]

Main idea: the longer the lifetime of an item, the more leeway we have in reporting it.
Leveled External-Memory Reporting Table (LERT) [Pandey ‘20]

• Given a stream of size $N$ and $\phi N > \Omega(N/M)$ the amortized cost of solving real-time event detection is

$$O \left( \left( \frac{1}{B} + \frac{1}{(\phi-1/M)N} \right) \log \frac{N}{M} \right)$$

• For a constant $\alpha$, can support arbitrarily small thresholds $\phi$ with amortized cost

$$O \left( \frac{1}{B} \log \frac{N}{M} \right)$$

**Takeaway**: Online reporting comes at the cost of throughput but almost online reporting is essentially free!
Given a stream of size $N$ and $\varphi N > \Omega(N/M)$ the amortized cost of solving real-time event detection is

$$O\left(\left(1 - \frac{1}{\varphi N}\right) \log N\right)$$

**Takeaway:** Online reporting comes at the cost of throughput but almost online reporting is essentially free!
Evaluation

- Empirical timeliness
- High-throughput ingestion
Evaluation: empirical time stretch

Average time stretch is 43% smaller than theoretical upper bound.
Evaluation: scalability

The insertion throughput increases as we add more threads. We can achieve > 13M insertions/sec.
LERT: supports scalable and real-time reporting

- Stream is large (e.g., terabytes) and high-speed (millions/sec)
- Events are high-consequence real-life events
- Malicious traffic forms a small portion of the stream

High throughput ingestion

No false-negatives; few false-positives

Timely reporting (real-time)

Very small reporting thresholds
Future work overview

Data Science

Scalable Data Systems

Data structures & Algorithms
Goal: Overcome *decades-old* data structure *trade-offs* using modern hardware and new algorithmic paradigms
Trade-off 1: Insertion throughput degrades with load factor

Insertion throughput vs load factor of state-of-the-art filters

Many update-intensive applications (e.g., network caches, data analytics, etc.) maintain filters at high load factors
Trade-off 1: Insertion throughput degrades with load factor

Performance suffers due to high-overhead of collision resolution
Combining techniques + new hardware

Combining hashing techniques (**Robin Hood + 2-choice hashing**)  
Using ultra-wide vector operations (**AVX512-BW**)
Combining techniques + new hardware

Combining hashing techniques (Robin Hood + 2-choice hashing)
Using ultra-wide vector operations (AVX512-BW)
Future work: Data Systems

Goal: Build a *population-scale* index on variation data to enable downstream apps gain *quick insights into variants*
Country-scale sequencing efforts produce huge amounts of sequencing data.

- 1000 Genomes project [https://www.internationalgenome.org/]
- The Cancer Genome Atlas (TCGA) [https://portal.gdc.cancer.gov/]
- Genotype-Tissue Expression (GTEx) [https://gtexportal.org/home/]
Variation data analysis can improve downstream applications

- Population-level disease analysis
- Genome-wide association studies
- Personalized medicine
- Cancer remission-rate prediction
- Colocalization analysis
- PCR primer design
- Genome assembly

**Sequencing & assembly**

- Count the number of variants in a gene
- List all people, with > N variants in a gene
- Return all positions with variants in a gene
- List all people, with sequence S in a gene
- For person P, return the closest variant from position X
Indexing in multiple coordinates is challenging

Reference-only indexes map positions only in the reference coordinate system

\[ f(p_i, p_j) \rightarrow (v_1 \ldots v_n), \text{ where } p_i \leq p_j \]

Pan-genome analysis involves queries based on sample coordinate systems

\[
\begin{align*}
\forall s & \in \{f_1, \ldots, f_S\} \\
\forall p_i, p_j & \in \mathbb{P}, p_i \leq p_j
\end{align*}
\]

Maintaining thousands of mappings increases computational complexity and memory footprint. Limits scalability to population-scale data.
Reference-only indexes map positions only in the reference coordinate system:

\[ f(p_i, p_j) \rightarrow (v_i \ldots v_n), \text{ where } p_i \leq p_j \]

Pan-genome analysis involves queries based on sample coordinate systems:

\[ f_s(p_i, p_j) \rightarrow (v_i \ldots v_n), \text{ where } p_i \leq p_j \]

Existing systems don’t support multiple coordinate systems. The ones that do, don’t scale beyond a few thousand samples.

Maintaining thousands of mappings increases computational complexity and memory footprint.

Limits scalability to population-scale data.
An inverted index on the pan-genome graph

- Partition the variation graph based on coordinate ranges
- Store partitions on disk
- Succinct index for reference coordinate system
- Local-graph exploration to map position from reference to sample coordinate

Queries often require loading 1-2 partitions
Future work: Data Science for genomics

Goal: Classification of metagenomic reads and identification of novel species using graph neural networks (GNN)
Metagenomic classification pipeline

Binning

Profiling

Classification to their species of origin, and an abundance profile

[Ye et al. 2019]
Existing techniques offer low recall

Classification is done based on the read contents
Use overlap relationship between reads

- Generate overlap graph: reads → nodes & overlap → edges
- Node features → Tetra nucleotide freq of reads
- Reference-based mapping as ground truth labels
Overlap graph + graph neural network (GNN)

Can achieve high recall using graph learning
Conclusion

- Scalability of data management systems will be the biggest challenge in future
- Changing hardware give rise to new algorithmic paradigms

We need to **redesign** existing data structures to take full advantage of modern hardware and **rebuild** data systems to efficiently support **future** data science.

https://prashantpandey.github.io
Quotient filter design

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]
Quotient filter design

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \]

\[ h(a) \]

<table>
<thead>
<tr>
<th>runends</th>
<th>occupieds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>( t(\alpha) )</td>
</tr>
</tbody>
</table>
Quotient filter design

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

\[ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \]

\[ \begin{array}{ccccccc}
\text{runends} & \text{occupieds} \\
\hline
& & & & & & \\
1 & & & & & & \\
& \downarrow & & & & & \\
& & t(a) & t(b) & & & \\
\end{array} \]
Quotient filter design

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

\[ \text{runends} \]
\[ \text{occupieds} \]

Abstract Representation

\[ 2^q \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
\end{array} \]

\[ t(a) \quad t(b) \quad t(d) \]
Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(a) )</td>
<td>( h(b) )</td>
<td>( h(d) )</td>
<td>( h(e) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| \( t(a) \) | \( t(b) \) | \( t(d) \) | \( t(e) \) |

Quotient filter design
Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]
Quotient filters can also be exact

- Quotient filters store $h(x)$ exactly
- To store $x$ exactly, use an invertible hash function

For $n$ elements and $p$-bit hash function:

Space usage: $\sim p \cdot \log_2 n$ bits/element
Cascade filter: write-optimized quotient filter
[Bender et al. ‘12, Pandey et al. ‘17]

- The Cascade filter efficiently scales out-of-RAM
- It accelerates insertions at some cost to queries
Cascade filter: flushing
[Bender et al. ‘12, Pandey et al. ‘17]

Items are initially inserted in the RAM level

Efficient merging

Quotient filter

log(N/M)

N

102
When RAM is full, items are flushed to the smallest level on disk $i$ with space to insert items in level 0 to $i-1$.
When RAM is full, items are flushed to the smallest level on disk $i$ with space to insert items in level 0 to $i-1$.
When RAM is full, items are flushed to the smallest level on disk \( i \) with space to insert items in level 0 to \( i-1 \)
Cascade filter: flushing
[Bender et al. ‘12, Pandey et al. ‘17]

When RAM is full, items are flushed to the smallest level on disk $i$ with space to insert items in level 0 to $i-1$
When RAM is full, items are flushed to the smallest level on disk $i$ with space to insert items in level 0 to $i-1$.
A query operation requires a lookup in each non-empty level
Time-stretch LERT

\[ \log\left(\frac{N}{M}\right) \]

Divide each level into \(1 + \frac{1}{\alpha}\), equal-sized bins.
Time-stretch LERT

When a bin is full, items move to the adjacent bin.
Time-stretch LERT

When a bin is full, items move to the adjacent bin.
Time-stretch LERT

\[ \log\left(\frac{N}{M}\right) \]

Quotient filter

Last bin **flushed** to first bin of the next level
Time-stretch LERT

While flushing consolidate counts; report if hits threshold

Last bin **flushed** to first bin of the next level
While flushing consolidate counts; report if hits threshold

Main idea: item is not put on a deeper level until it’s “aged sufficiently”

Last bin flusheds to first bin of the next level
Time-stretch LERT I/O complexity

\[ O \left( \left( \frac{\alpha + 1}{\alpha} \right) \frac{1}{B} \log \frac{N}{M} \right) \]

Optimal insert cost for Write-optimized data structure
Time-stretch LERT I/O complexity

\[ O \left( \left( \frac{\alpha + 1}{\alpha} \right) \frac{1}{B} \log \frac{N}{M} \right) \]

Extra cost because we only move one bin during a flush. Constant loss for constant \( \alpha \)

Optimal insert cost for Write-optimized data structure
Trade-off 2: “One-size-fits-all” approach leaves performance on table

LIGRA [Shun & Blelloch ‘13]

<table>
<thead>
<tr>
<th>Operation</th>
<th>LIGRA</th>
<th>ASPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_edge</code></td>
<td>$O((</td>
<td>E</td>
</tr>
<tr>
<td><code>get_neighbors</code></td>
<td>$O(deg(u)/B)$</td>
<td>$O(\log</td>
</tr>
</tbody>
</table>

Neighbor access requires at least **two cache misses**

For dynamic, all operations have a **log factor**
Trade-off 2: “One-size-fits-all” approach leaves performance on table

### Static → Fast computations; no updates
### Dynamic → Slower computations; updates

<table>
<thead>
<tr>
<th>Operation</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_edge</td>
<td>$O((</td>
<td>E</td>
</tr>
<tr>
<td>get_neighbors</td>
<td>$O(\text{deg}(u)/B)$</td>
<td>$O(\log</td>
</tr>
</tbody>
</table>

Neighbor access requires at least **two cache misses**
For dynamic, all operations have a **log factor**
Real world graphs are often skewed

- Dynamic partitioning of vertices based on the degree
- Separate structures for each partition to minimize cache misses

High variance in the degree distribution
Dynamic partitioning + hierarchical structure

- Dynamic partitioning of vertices based on the degree
- Separate structures for each partition to minimize cache misses

High variance in the degree distribution

![Graph showing throughput vs. batch size](image)

![Normalized running time comparison](image)
Dynamic partitioning + hierarchical structure

- Dynamic partitioning of vertices based on the degree
- Separate structures for each partition to minimize cache misses

High variance in the degree distribution

Terrace: Fast updates

Terrace: Faster computations

Graph showing throughput vs. batch size:
- Terrace Insert LJ
- Terrace Insert Orkut
- Aspen Insert LJ
- Aspen Insert Orkut

Bar chart showing normalized running time for different algorithms:
- BFS
- PR
- BC
- CC
- SSSP
- TC
My goal as a researcher is to build **scalable data systems** to **accelerate** and **scale data science** applications.
Our contribution

Combine streaming and EM algorithms to solve real-time event detection problem