Reconfigurable Inverted Index

Yusuke Matsui\textsuperscript{1}  Ryota Hinami\textsuperscript{2}  Shin’ichi Satoh\textsuperscript{1}

\textsuperscript{1}National Institute of Informatics  \textsuperscript{2}The University of Tokyo

slides: https://bit.ly/2P0KuW1
Approximate nearest neighbor search

$$q \in \mathbb{R}^D$$

ANN system
hash-table, trees, inverted-index, etc

$$\text{argmin}_{n \in \{1, \ldots, N\}} \| q - x_n \|^2$$

Result
$$\begin{bmatrix} 0.20 \\ 3.25 \\ 0.72 \\ 1.68 \end{bmatrix}$$

Add

Database vectors
$$\begin{bmatrix} 5.22 & 4.63 & 0.86 \\ 0.54 & 6.21 & 3.44 \\ 1.66 & 0.72 & 1.12 \\ 0.74 & 0.31 & 0.04 \end{bmatrix}$$

$$x_1 \quad x_2 \quad x_N$$

slides: https://bit.ly/2P0KuW1
Approximate nearest neighbor search

\[ q \in \mathbb{R}^D \]

**Query**

**ANN system**

hash-table, trees, inverted-index, etc

**Database vectors**

\[
\begin{bmatrix}
5.22 & 4.63 \\
0.54 & 6.21 \\
1.66 & 0.72 \\
0.74 & 0.31 \\
\end{bmatrix}
\]

**Add**

**Approximate NN search**

\[
\text{argmin}_{n \in \{1, \ldots, N\}} \| q - x_n \|_2^2
\]

**Result**

\[
\begin{bmatrix}
0.20 \\
3.25 \\
0.72 \\
1.68 \\
\end{bmatrix}
\]

\[
x_{74}
\]

slides: https://bit.ly/2P0KuW1
Approximate nearest neighbor search

\[ q \in \mathbb{R}^D \]

ANN system

hash-table, trees, inverted-index, etc

\[ \text{argmin}_{n \in \{1, \ldots, N\}} \| q - x_n \|_2^2 \]

Result

\[ \begin{bmatrix} 0.20 \\ 3.25 \\ 0.72 \\ 1.68 \end{bmatrix} \]

Database vectors

\[ \begin{bmatrix} 5.22 & 4.63 & 0.86 \\ 0.54 & 6.21 & 3.44 \\ 1.66 & 0.72 & 1.12 \\ 0.74 & 0.31 & 0.04 \end{bmatrix} \]

Add

\[ x_1 \quad x_2 \quad \cdots \quad x_N \]

slides: https://bit.ly/2P0KuW1
Related work

➢ Locality-sensitive-hashing (LSH)
  - FALCONN [Andoni+, 15] [Razenshteyn+, 18]

➢ Project/tree-based
  - FLANN [Muja+, 14]
  - Annoy [Bernhardsson, 18]

➢ Graph traversal
  - NSW/HNSW on NMSLIB [Malkov+, 16] [Boytsov+, 13]

➢ Product quantization (PQ)
  - IVFPQ on Faiss [Jégou+, 11] [Johnson+, 17] etc.
  - Our Reconfigurable Inverted Index
Subset search problem

- Existing ANN systems are fast for the all vectors
  - Search is over $S = \{1, \ldots, N\}$

- However, it is hard to run the search for a subset
  - Search is over $S \subseteq \{1, \ldots, N\}$
  - e.g., searching from $\{x_{1000}, \ldots, x_{2000}\}$
  - Why? Systems are usually optimized for $S = \{1, \ldots, N\}$
There is a demand for subset search!

slides: https://bit.ly/2P0KuW1
There is a demand for subset search!

Propose: Reconfigurable inverted index (Rii)
✓ Subset search
✓ A comparative performance with IVFPQ (Faiss)
✓ 10 ms for billion-scale data
Reconfigurable inverted index (Rii)

➢ Preliminary
  - PQ linear scan
  - IVFPQ

➢ Data structure

➢ Search

<table>
<thead>
<tr>
<th>( \mathcal{S} ) Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast if (</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \mathcal{S} ) Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast if (</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \mathcal{S} ) Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cherry pick! Always fast</td>
</tr>
</tbody>
</table>
Reconfigurable inverted index (Rii)

➢ Preliminary
  - PQ linear scan
  - IVFPQ

➢ Data structure

➢ Search

Fast if $|S|$ is small

Fast if $|S|$ is large

Cherry pick!
Always fast
Preliminary: Product quantization (PQ) [Jégou+, TPAMI 11]

**PQ:** Compress a vector into a short code

\[
\begin{bmatrix}
5.22 \\
0.54 \\
1.66 \\
0.74
\end{bmatrix}
\]

\(\mathbb{R}^4 \rightarrow \{\text{ }\},\ldots\}^2\)

All database vectors are PQ-encoded beforehand

\[
\begin{bmatrix}
x_1 \\
x_2 \\
x_N
\end{bmatrix}
\]

\[
\begin{bmatrix}
5.22 & 4.63 \\
0.54 & 6.21 \\
1.66 & 0.72 \\
0.74 & 0.31
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.86 \\
3.44 \\
1.12 \\
0.04
\end{bmatrix}
\]

slides: https://bit.ly/2P0KuW1
Preliminary: Product quantization (PQ) [Jégou+, TPAMI 11]

- The subset search is possible with a linear cost of $|S|$

\[
\text{argmin}_{n \in S} d(q, n)
\]

- The search is efficient only if $|S|$ is small

\[q \in \mathbb{R}^D\]

\[
\begin{bmatrix}
0.23 \\
3.15 \\
0.65 \\
1.43
\end{bmatrix}
\]

\[\mathcal{S} = \{2, 4, 5, 8\}\]

Runtime

\[N\]

| $|S|$ | 1 | 2 | 3 | 4 | 5 | 6 | ... |
|------|---|---|---|---|---|---|------|

slides: https://bit.ly/2P0KuW1
Reconfigurable inverted index (Rii)

➢ Preliminary
  - PQ linear scan
  - IVFPQ

➢ Data structure

➢ Search

➢ Evaluation

Fast if $|S|$ is small
Fast if $|S|$ is large
Cherry pick! Always fast

slides: https://bit.ly/2P0KuW1
Preliminary: Inverted Index + PQ (IVFPQ) [Jégou+, TPAMI 11]

➢ Current basic data structure for a large-scale search
➢ Subset-search is possible only if $|S|$ is large

Space partitioning

slides: https://bit.ly/2P0KuW1
Preliminary: Inverted Index + PQ (IVFPQ) [Jégou+, TPAMI 11]

- Current basic data structure for a large-scale search
- Subset-search is possible only if $|S|$ is large

Space partitioning

1. Find the closest space: $k^* = \arg\min_k \|q - c_k\|^2$
2. Focus the $k^*$th space, accept items $\in S$
3. Re-rank the items via PQ-linear scan

Examples:
- $S = \{13, 92, 105, \ldots\}$
- $n \in S$ or not

Re-rank via PQ-linear scan

slides: https://bit.ly/2P0KuW1
Preliminary: Inverted Index + PQ (IVFPQ) [Jégou+, TPAMI 11]

➢ Current basic data structure for a large-scale search
➢ Subset-search is possible only if $|S|$ is large

Why is it slow for small $|S|$?

1. Find the closest space: $k^* = \arg\min_k q - c_k$
2. Focus the $k^*$th space, accept items $\in S$
3. Re-rank the items via PQ-linear scan

Re-rank via PQ-linear scan

Why is it slow for small $|S|$?

e.g., if $|S|$ is small and they are far away from the query, we might need to scan all items $\in S$

$\mathbf{c} \in \mathbb{R}^D$ Re-rank via PQ-linear scan

$n \in S$ or not

e.g., $S = \{13, 92, 105, \ldots\}$

Runtime

$N$
Reconfigurable inverted index (Rii)

➢ Preliminary
  - PQ linear scan
  - IVFPQ

➢ Data structure

➢ Search

Runtime vs. $|S|$:
- Fast if $|S|$ is small
- Fast if $|S|$ is large
- Cherry pick! Always fast

slides: https://bit.ly/2P0KuW1
Data structure

➢ Store (1) PQ-codes **linearly**, and (2) IDs as an inverted index
➢ Can run either PQ-linear-scan or IVFPQ with a **single data structure**

Key: store codes linearly

cf. IVFPQ

➢ PQ-codes are also chunked. Natural
➢ Slight, but critical change
Reconfigurable inverted index (Rii)

➢ Preliminary
   - PQ linear scan
   - IVFPQ

➢ Data structure

➢ Search

Runtime vs. \(|S|\)

Fast if \(|S|\) is small

Fast if \(|S|\) is large

Cherry pick! Always fast

slides: https://bit.ly/2P0KuW1
Search

➢ If $|S|$ is small, run PQ-linear scan
➢ If $|S|$ is large, run IVFPQ

$q \in \mathbb{R}^D$

Runtime

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Search

➢ If $|S|$ is small, run PQ-linear scan
➢ If $|S|$ is large, run IVFPQ

$\mathcal{S}$

$q \in \mathbb{R}^D$

Linearly compared

Runtime vs $|S|$:
- Blue line: PQ-linear scan
- Orange line: IVFPQ

slides: https://bit.ly/2P0KuW1
If $|S|$ is small, run PQ-linear scan

If $|S|$ is large, run IVFPQ

slides: https://bit.ly/2P0KuW1
If $|S|$ is small, run PQ-linear scan

If $|S|$ is large, run IVFPQ

If $|S|$ is small, run PQ-linear scan

If $|S|$ is large, run IVFPQ
If $|S|$ is small, run PQ-linear scan
If $|S|$ is large, run IVFPQ

Set a threshold $\theta$
Key: Switch two methods based on $|S| \leq \theta$

Search

- If $|S|$ is small, run PQ-linear scan
- If $|S|$ is large, run IVFPQ

Use PQ-linear-scan
Use IVFPQ

slides: https://bit.ly/2P0KuW1
Search

➢ If \( |S| \) is small, run PQ-linear scan
➢ If \( |S| \) is large, run IVFPQ

Set a threshold \( \theta \)
Key: Switch two methods based on \( |S| \leq \theta \)

Runtime

\[ \begin{array}{c|c|c|c}
\hline \hline
N & 0.65 & 3.15 & 0.23 \\
\hline \end{array} \]

\( q \in \mathbb{R}^D \)

\( c_1, c_2, c_5 \)

\( \theta \)

\( \Theta \)

\[ \begin{array}{c|c|c|c}
\hline \hline
10^3 & 10^4 & 10^5 \\
\hline \end{array} \]

PQ linear scan (Alg. 1)
Inverted index (Alg. 2)
Final query (Alg. 3)

slides: https://bit.ly/2P0KuW1
Evaluation

➢ SIFT1M ($N = 10^6, D = 128$). Results for top-R search
Evaluation

SIFT1M ($N = 10^6, D = 128$). Results for top-$R$ search

- Existing system: Annoy
- Force to search a subset

The existing system is slow, especially when $|S|$ is small

Proposed Rii is always fast regardless of $|S|$ and $R$
import rii
import nanopq

# Prepare a PQ/OPQ codec with M=32 sub spaces
codec = nanopq.PQ(M=32).fit(vecs=Xt)  # Trained using Xt

# Instantiate a Rii class with the codec
e = rii.Rii(fine_quantizer=codec)

# Add vectors
e.add_configure(vecs=X)

# Search
ids, dists = e.query(q=q, topk=3, target_ids=S)
print(ids, dists)  # e.g., [7484 8173 1556] [15.062 15.385 16.169]
Summary

Approximate NN Search

\[
\arg\min_{n \in S} \| q - x_n \|_2^2
\]

Reconfigurable inverted index:
- Store PQ-codes linearly
- Switch method based on \(|S|\)

Runtime

Use PQ-linear-scan

Use IVFPQ

PyPI:

\[
\$ \text{pip install rii}
\]

One thing I couldn’t mention:

\textbf{Reconfiguration}: the system remains fast even after many new items are added

See our paper, or come to our poster:

✓ Poster session 5 (15:30 – 16:30)
Example: Image search

\[
\begin{bmatrix}
0.23 \\
3.15 \\
0.65 \\
1.43
\end{bmatrix}
\]

\( q \in \mathbb{R}^D \)

Extract a VGG feature

ANN system

\[
\text{argmin}_{n \in \{1, \ldots, N\}} \| q - x_n \|_2^2
\]

Result

\[
\begin{bmatrix}
0.20 \\
3.25 \\
0.72 \\
1.68
\end{bmatrix}
\]

\( x_{74} \)

74th image

Slides: https://bit.ly/2P0KuW1
**Example of subset search for image**

![Diagram of subset search](slides: https://bit.ly/2P0KuW1)

Let's say we have a query image represented by a vector $q = \begin{bmatrix} 0.23 \\ 3.15 \\ 0.65 \\ 1.43 \end{bmatrix}$ belonging to $\mathbb{R}^D$.

We perform an approximate nearest neighbor (ANN) search using an ANN system. The search is given by:

$$\text{argmin}_{n \in S} \| q - x_n \|_2^2$$

The result is $x_{92}$ with the following coordinates:

$$\begin{bmatrix} 0.20 \\ 3.25 \\ 0.72 \\ 1.68 \end{bmatrix}$$

Filtering by shooting-date:
- Taken in 2018
- $S = \{34, 56, 92, \ldots, 663\}$

**Query image**: A dog.

**N database images**: Various images including a tower, a woman, a bird, a mask, and clothing items.

**92nd image**: A dog similar to the query image.
Evaluation

➢ Extensive comparison against existing methods
➢ For a fixed accuracy (Recall@1), check runtime and its disk space

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Parameters</th>
<th>Recall@1 (fixed)</th>
<th>Runtime/query</th>
<th>Disk space</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT1M</td>
<td>Annoy [11]</td>
<td>$n_{trees} = 2000$, $k_{search} = 400$</td>
<td>0.67</td>
<td>0.18 ms</td>
<td>1703 MB</td>
<td>899 sec</td>
</tr>
<tr>
<td></td>
<td>FALCONN [1, 41]</td>
<td>$n_{probes} = 16$</td>
<td>0.63</td>
<td>0.87 ms</td>
<td>-</td>
<td>1.8 sec</td>
</tr>
<tr>
<td></td>
<td>NMSLIB (HNSW) [14, 33, 39]</td>
<td>$efS = 4$</td>
<td>0.67</td>
<td><strong>0.043 ms</strong></td>
<td>669 MB</td>
<td>436 sec</td>
</tr>
<tr>
<td></td>
<td>Faiss (IVFADC) [25, 26]</td>
<td>$K = 10^3, M = 64, n_{probe} = 4$</td>
<td>0.67</td>
<td>0.61 ms</td>
<td>73 MB</td>
<td>30 sec</td>
</tr>
<tr>
<td></td>
<td>Rii (proposed)</td>
<td>$K = 10^3, M = 64, L = 5000$</td>
<td>0.64</td>
<td>0.73 ms</td>
<td><strong>69 MB</strong></td>
<td>82 sec</td>
</tr>
<tr>
<td></td>
<td>Rii-OPQ (proposed)</td>
<td>$K = 10^3, M = 64, L = 5000$</td>
<td>0.65</td>
<td>0.82 ms</td>
<td><strong>69 MB</strong></td>
<td>85 sec</td>
</tr>
<tr>
<td>GIST1M</td>
<td>Annoy [11]</td>
<td>$n_{trees} = 2000$, $k_{search} = 2000$</td>
<td>0.49</td>
<td>1.2 ms</td>
<td>5023 MB</td>
<td>2088 sec</td>
</tr>
<tr>
<td></td>
<td>FALCONN [1, 41]</td>
<td>$n_{probes} = 512$</td>
<td>0.53</td>
<td>8.6 ms</td>
<td>-</td>
<td>7.2 sec</td>
</tr>
<tr>
<td></td>
<td>NMSLIB (HNSW) [14, 33, 39]</td>
<td>$efS = 8$</td>
<td>0.49</td>
<td><strong>0.19 ms</strong></td>
<td>3997 MB</td>
<td>1576 sec</td>
</tr>
<tr>
<td></td>
<td>Faiss (IVFADC) [25, 26]</td>
<td>$K = 10^3, M = 240, n_{probe} = 8$</td>
<td>0.52</td>
<td>3.8 ms</td>
<td>253 MB</td>
<td>51 sec</td>
</tr>
<tr>
<td></td>
<td>Rii (proposed)</td>
<td>$K = 10^3, M = 240, L = 8000$</td>
<td>0.45</td>
<td>3.2 ms</td>
<td><strong>246 MB</strong></td>
<td>353 sec</td>
</tr>
<tr>
<td></td>
<td>Rii-OPQ (proposed)</td>
<td>$K = 10^3, M = 240, L = 8000$</td>
<td>0.50</td>
<td>3.8 ms</td>
<td><strong>249 MB</strong></td>
<td>388 sec</td>
</tr>
</tbody>
</table>
**Evaluation**

- Extensive comparison against existing methods
- For a fixed accuracy (Recall@1), check runtime and its disk space

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Parameters</th>
<th>Recall@1 (fixed)</th>
<th>Runtime/query</th>
<th>Disk space</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT1M</td>
<td>Annoy [11]</td>
<td>$n_{trees} = 2000, k_{search} = 400$</td>
<td>0.67</td>
<td>0.18 ms</td>
<td>1703 MB</td>
<td>899 sec</td>
</tr>
<tr>
<td></td>
<td>FALCONN [1, 41]</td>
<td>$n_{probes} = 16$</td>
<td>0.63</td>
<td>0.87 ms</td>
<td>-</td>
<td>1.8 sec</td>
</tr>
<tr>
<td></td>
<td>NMSLIB (HNSW) [14, 33, 39]</td>
<td>efS = 4</td>
<td>0.67</td>
<td><strong>0.043 ms</strong></td>
<td>669 MB</td>
<td>436 sec</td>
</tr>
<tr>
<td></td>
<td>Faiss (IVFADC) [25, 26]</td>
<td>$K = 10^3, M = 64, n_{probe} = 4$</td>
<td>0.67</td>
<td>0.61 ms</td>
<td>73 MB</td>
<td>30 sec</td>
</tr>
<tr>
<td></td>
<td>Rii (proposed)</td>
<td>$K = 10^3, M = 5000$</td>
<td>0.64</td>
<td>0.73 ms</td>
<td><strong>69 MB</strong></td>
<td>82 sec</td>
</tr>
<tr>
<td></td>
<td>Rii-OPQ (proposed)</td>
<td>$K = 10^3, M = 5000$</td>
<td>0.65</td>
<td>0.82 ms</td>
<td><strong>69 MB</strong></td>
<td>85 sec</td>
</tr>
</tbody>
</table>

NMSLIB is extremely fast, but consume relatively large disk space (∼memory)
## Evaluation

- Extensive comparison against existing methods
- For a fixed accuracy (Recall@1), check runtime and its disk space

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Parameters</th>
<th>Recall@1 (fixed)</th>
<th>Runtime/query</th>
<th>Disk space</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT1M</td>
<td>Annoy [11]</td>
<td>(n_{\text{trees}} = 2000, k_{\text{search}} = 400)</td>
<td>0.67</td>
<td>0.18 ms</td>
<td>1703 MB</td>
<td>899 sec</td>
</tr>
<tr>
<td></td>
<td>FALCONN [1, 41]</td>
<td>(n_{\text{probes}} = 16)</td>
<td>0.63</td>
<td>0.87 ms</td>
<td>-</td>
<td>1.8 sec</td>
</tr>
<tr>
<td></td>
<td>NMSLIB (HNSW) [14, 33, 39]</td>
<td>(efS = 4)</td>
<td>0.67</td>
<td>0.043 ms</td>
<td>669 MB</td>
<td>436 sec</td>
</tr>
<tr>
<td></td>
<td>Faiss (IVFADC) [25, 26]</td>
<td>(K = 10^3, M = 64, \text{probe} = 4)</td>
<td>0.67</td>
<td>0.61 ms</td>
<td>73 MB</td>
<td>30 sec</td>
</tr>
<tr>
<td></td>
<td>Rii (proposed)</td>
<td>(K = 10^3, M = 5000)</td>
<td>0.64</td>
<td>0.73 ms</td>
<td>69 MB</td>
<td>82 sec</td>
</tr>
<tr>
<td></td>
<td>Rii-OPQ (proposed)</td>
<td>(K = 10^3, M = 5000)</td>
<td>0.65</td>
<td>0.82 ms</td>
<td>69 MB</td>
<td>85 sec</td>
</tr>
</tbody>
</table>

NMSLIB is extremely fast, but consume relatively large disk space (∼memory)

Proposed Rii achieved a comparative performance with Faiss

slides: https://bit.ly/2P0KuW1