Building a Scalable Multimedia Search Engine Using Infiniband

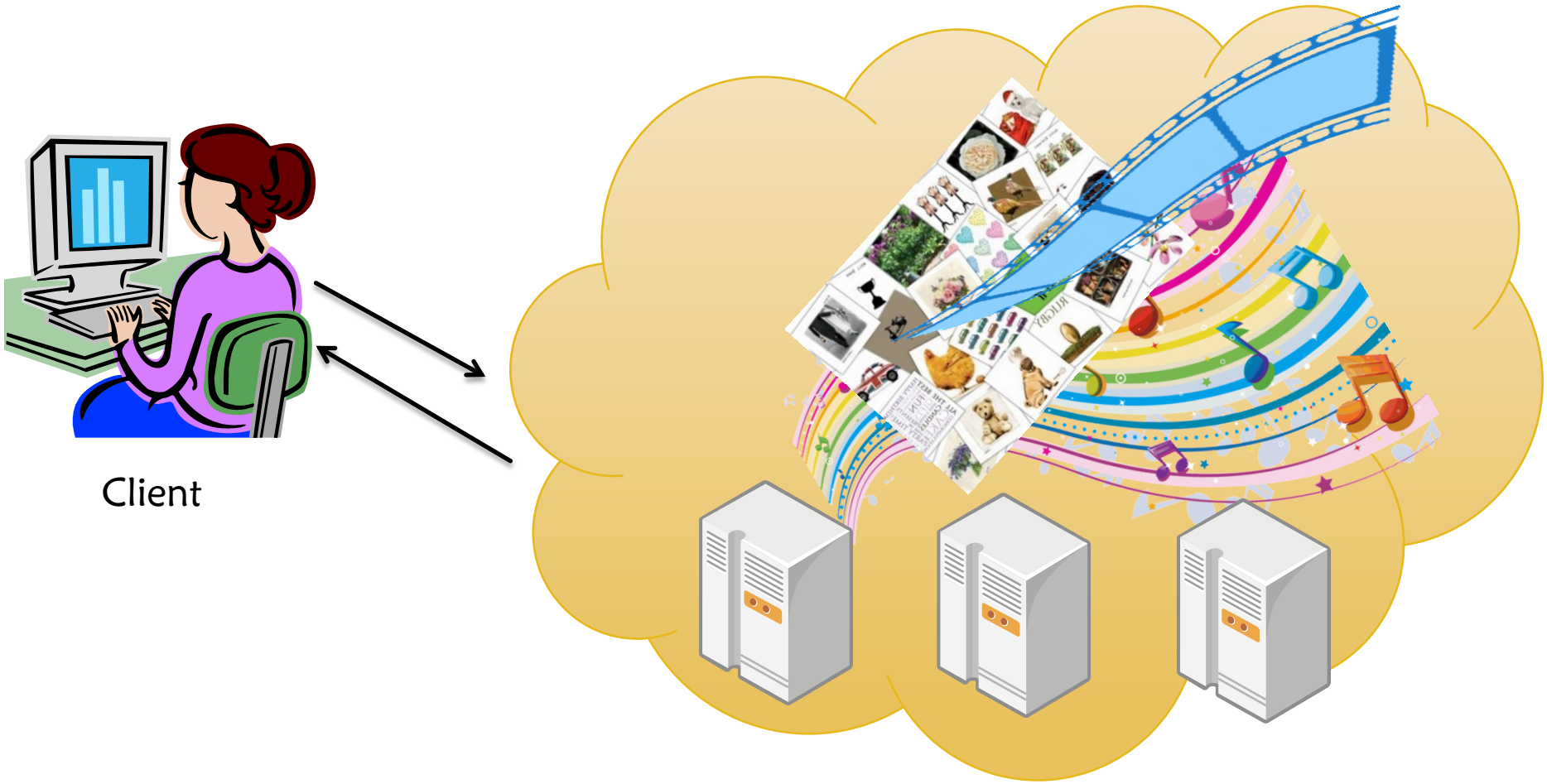
Qi Chen

Yisheng Liao, Christopher Mitchell, Jinyang Li, Zhen Xiao

Peking University

NYU

Online search must be scalable



Client

Example Search Engines: Google, Bing

How multimedia search is done

billions of Features

billions of images

	Feature 1	Feature 2	…	Feature x
Img 1	1	1		
Img 2		1		1
…				
…				
Img n		1		1

Indexing
(usually done offline)

How multimedia search is done

billions of Features

billions of images

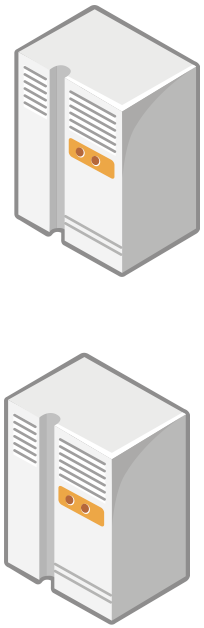
	Feature 1	Feature 2	…	Feature x
Img 1	1	1		
Img 2		1		1
…				
…				
Img n		1		1

Query image

Search features
$f_1, f_2, \cdots, f_n$

Indexing
(usually done offline)

Two ways to distribute: horizontal partition



	Feature 1	Feature 2	…	Feature f
Img 1	1	1		
Img 2		1		1
Img 3			1	
…				
…				
…				
…				
Img n		1		1

Not scalable because a query must contact all servers

Two ways to distribute: vertical partition

	Feature 1	Feature 2	…	…	Feature f
Img 1	1	1			
Img 2		1			1
…					
…					
Img n		1			1

Expensive because a query may look up tens of thousands of features

Horizontal vs. vertical: State-of-art and new opportunity

- Horizontal beats vertical partitioning on the Ethernet
- But..
- Ultra-low latency network is coming to data centers
  - Infiniband, RoCE
  - RTT ≈ 10us. (compared to RTT >100us on Ethernet)
- Our insight: Vertical beats horizontal on low-latency networks
  - Why latency matters: Use more roundtrips to reduce feature lookups

Outlines

1. Motivation
2. VertiCut Design
3. Evaluation
4. Related Work

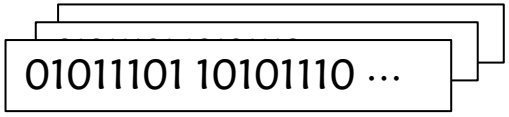
Overview of VertiCut Image Search

- Indexing
  - Offline indexer transforms images to 128-bit binary codes

feature extraction using SIFT

Collection of 128-dim vectors

binary transformation using Locality Sensitive Hashing (LSH)



Collection of 128-bit binary codes

- Searching
  - Online k-nearest-nbr (KNN) algorithm finds k codes with smallest hamming distance to a query code

How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

Code(x) = **011…111 000…101 000…101 001…110**

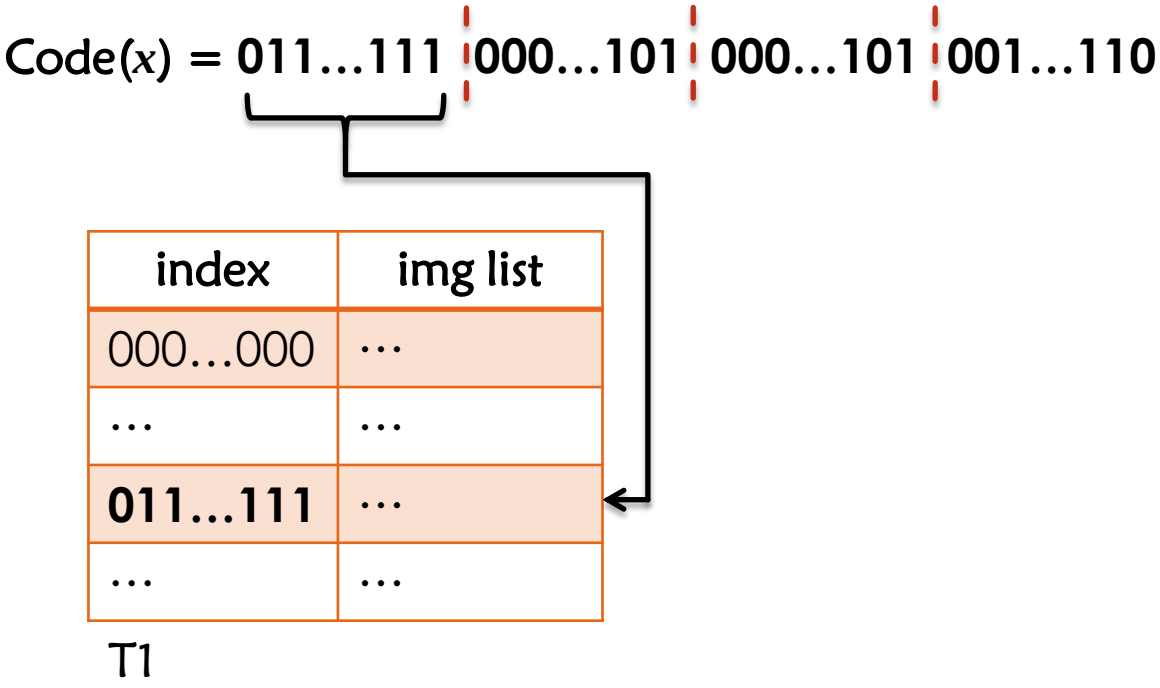
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

Code(*x*) = **011…111 | 000…101 | 000…101 | 001…110**

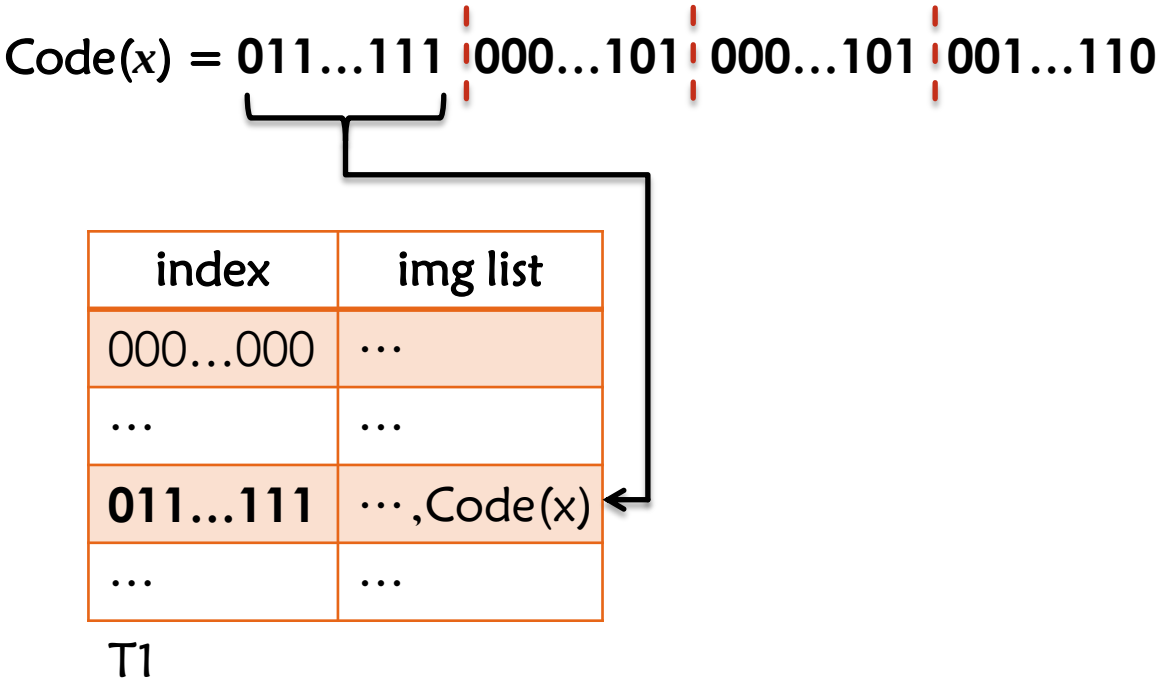
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti



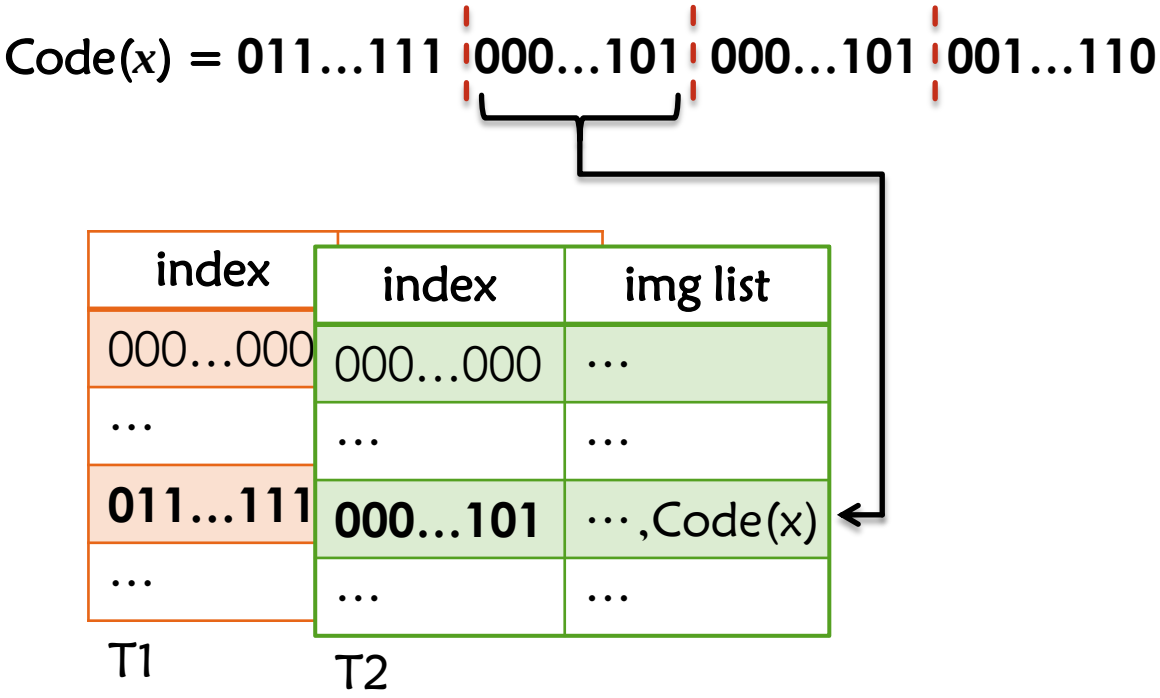
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti



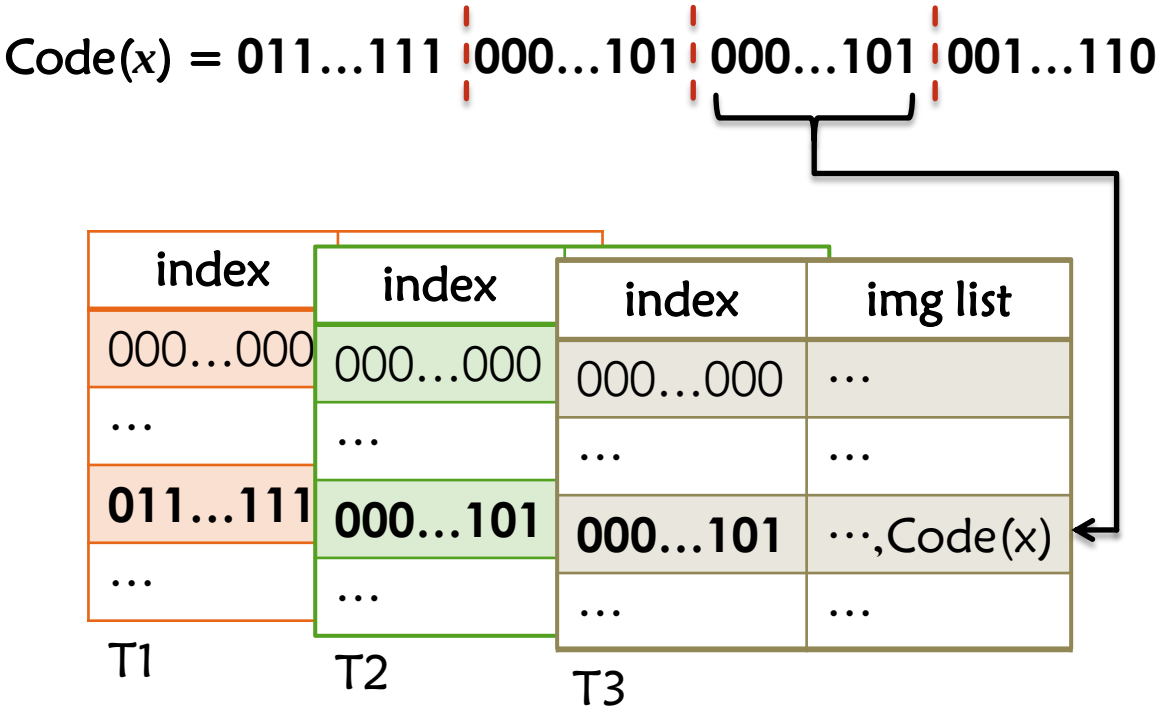
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti



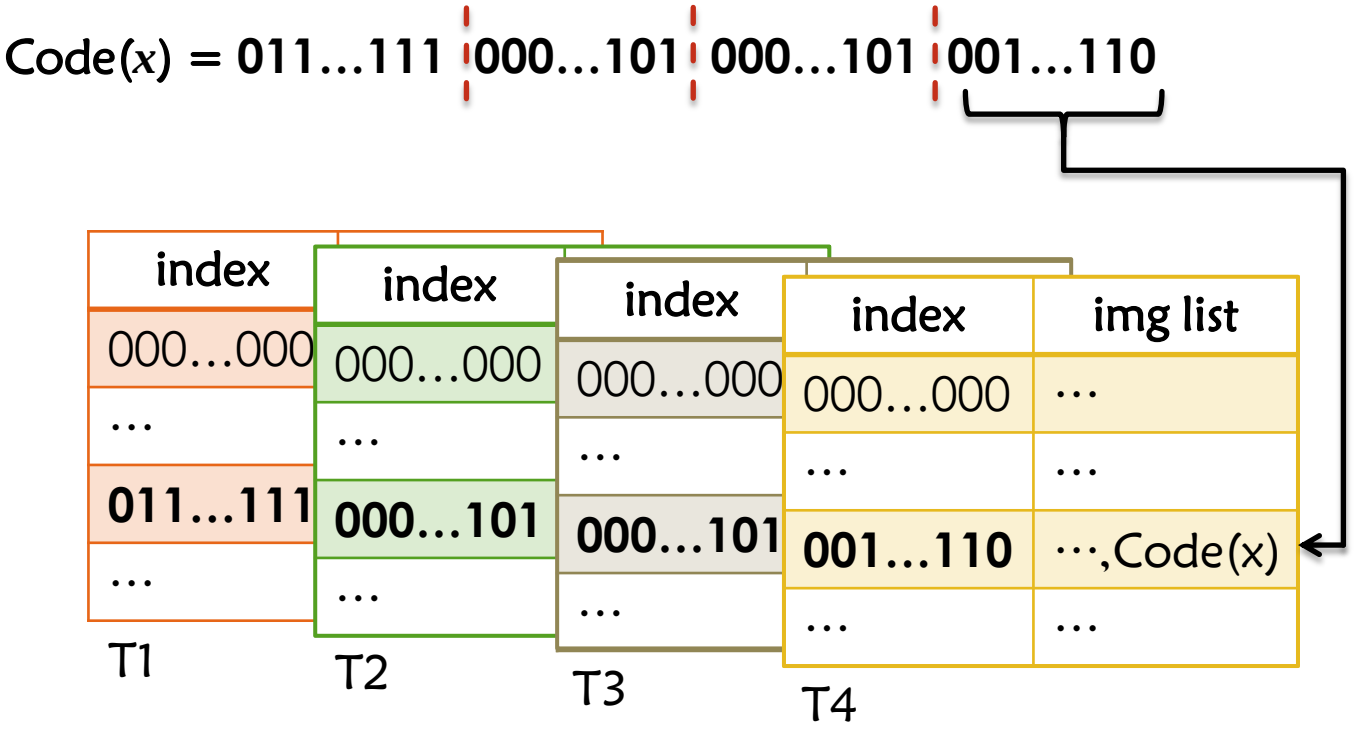
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

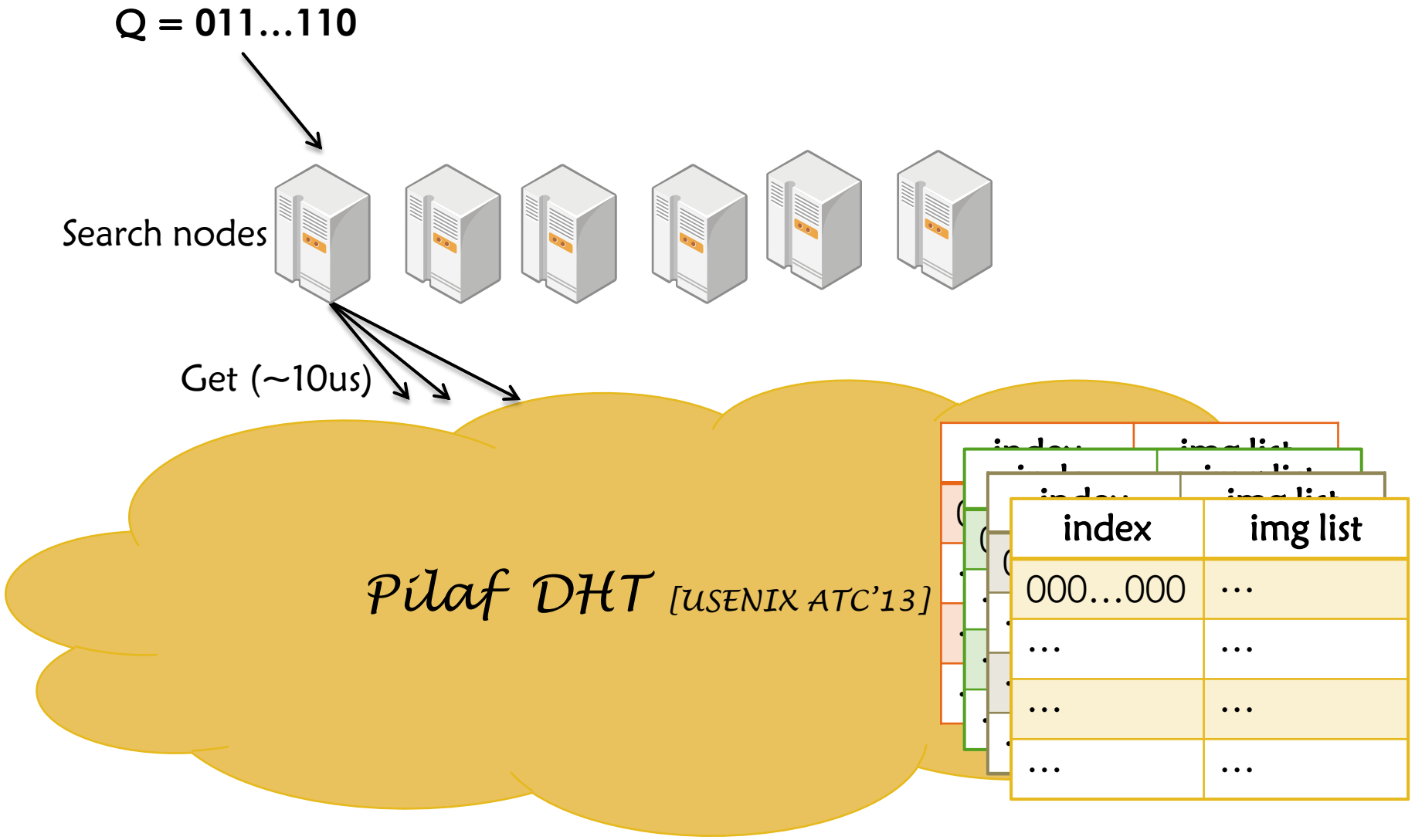


How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR'12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

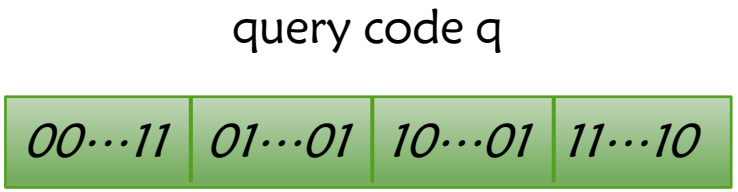


VertiCut search architecture



How to do KNN in binary space?

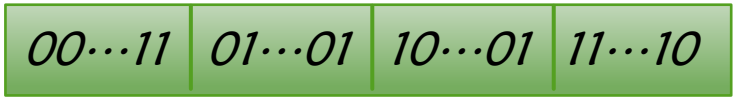
- To search 100 KNNs given a query code q



How to do KNN in binary space?

- To search 100 KNNs given a query code q

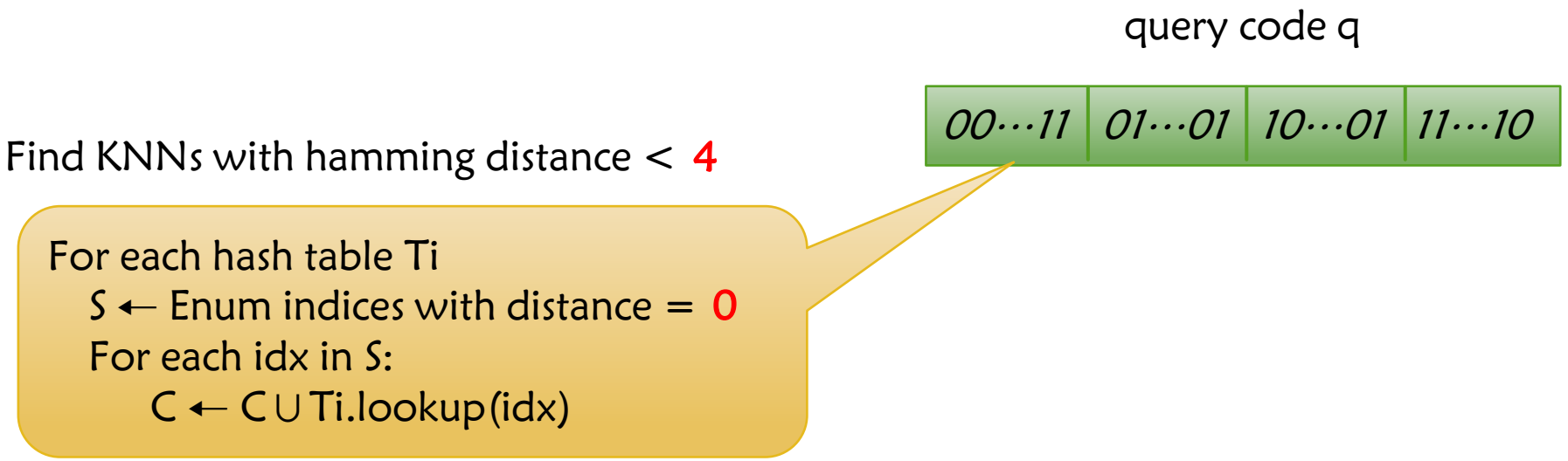
query code q



Find KNNs with hamming distance < 4

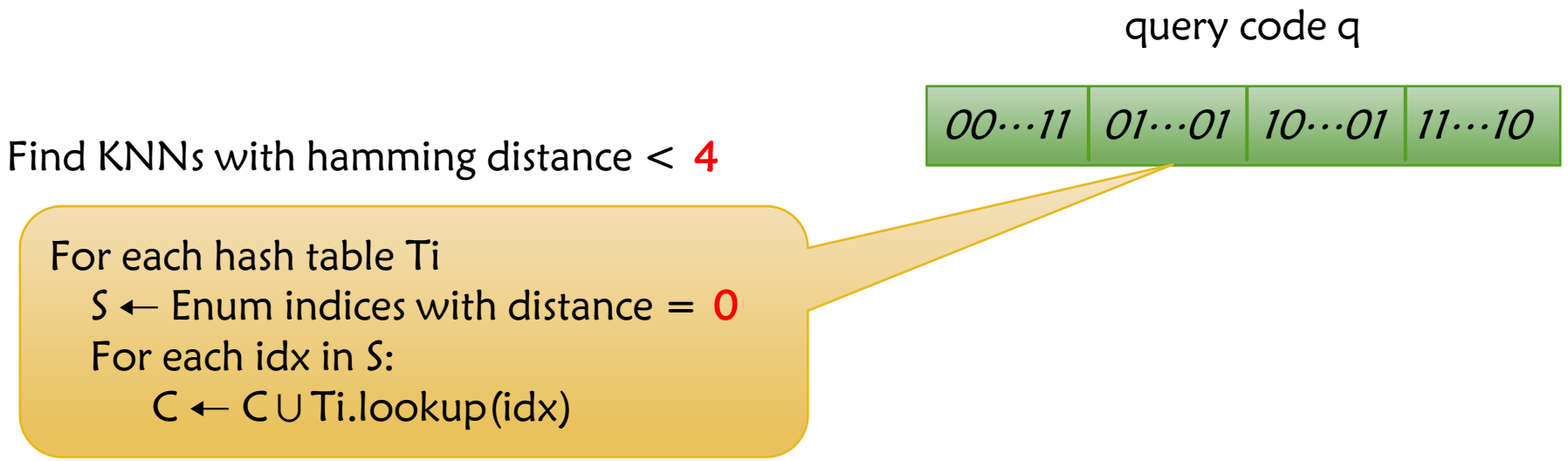
How to do KNN in binary space?

- To search 100 KNNs given a query code q



How to do KNN in binary space?

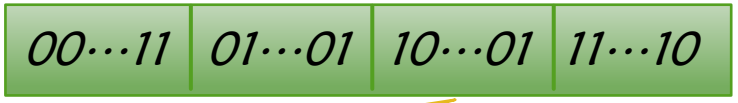
- To search 100 KNNs given a query code q



How to do KNN in binary space?

- To search 100 KNNs given a query code q

query code q

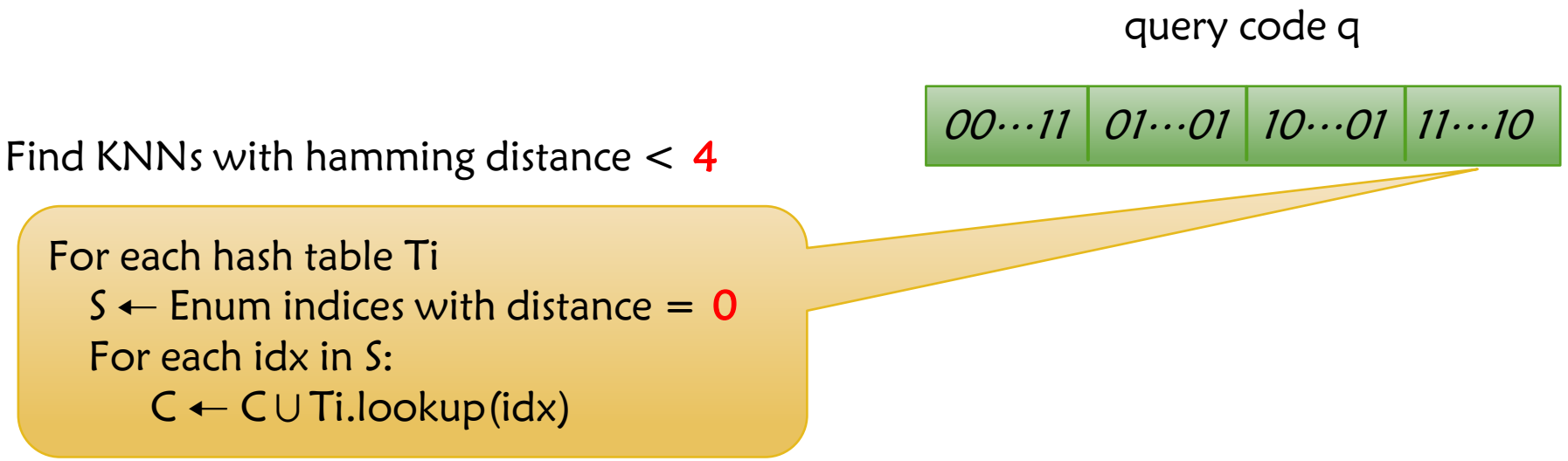


Find KNNs with hamming distance < 4

For each hash table Ti
  S ← Enum indices with distance = 0
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)

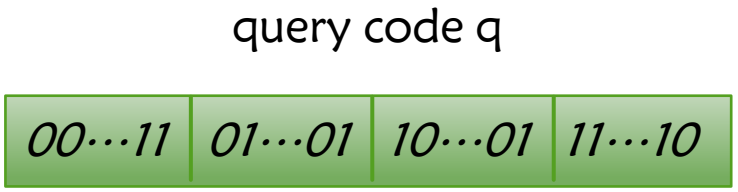
How to do KNN in binary space?

- To search 100 KNNs given a query code q



How to do KNN in binary space?

- To search 100 KNNs given a query code q



Find KNNs with hamming distance < 4

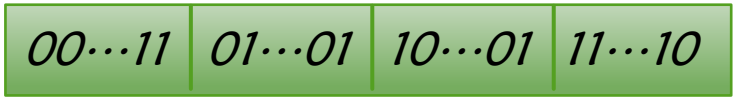
```
For each hash table Ti
    S ← Enum indices with distance = 0
    For each idx in S:
        C ← C ∪ Ti.lookup(idx)
```

```
For each image code x in C:
    if distance(x, q) < 4 :
        add x to result
if |result| >= 100:
    return KNN in result
```

How to do KNN in binary space?

- To search 100 KNNs given a query code q

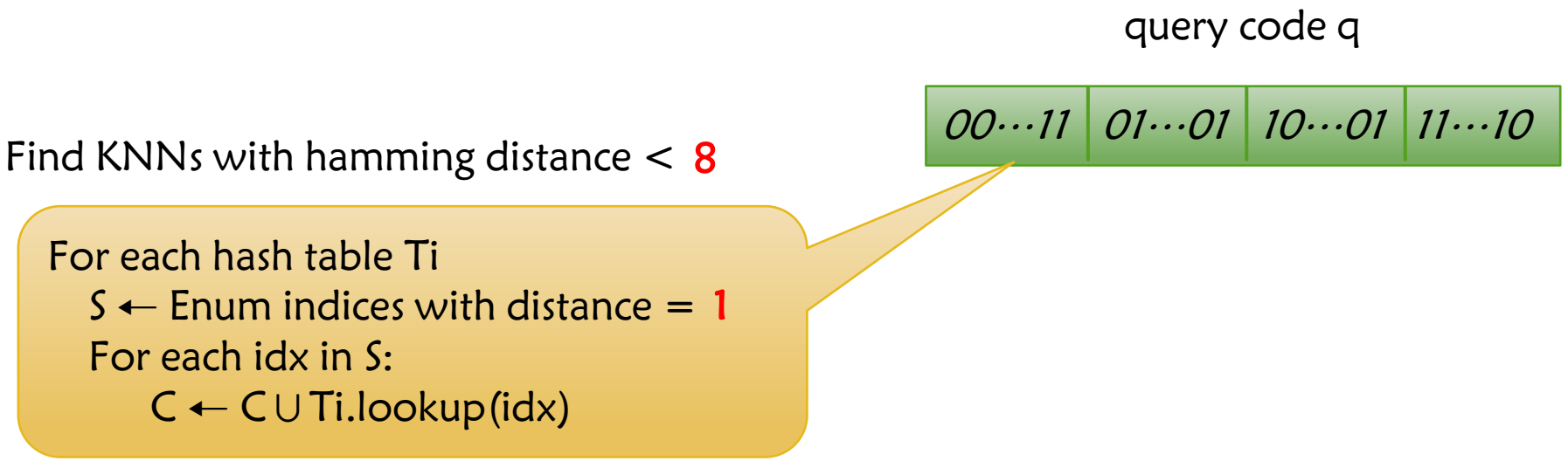
query code q



Find KNNs with hamming distance < 8

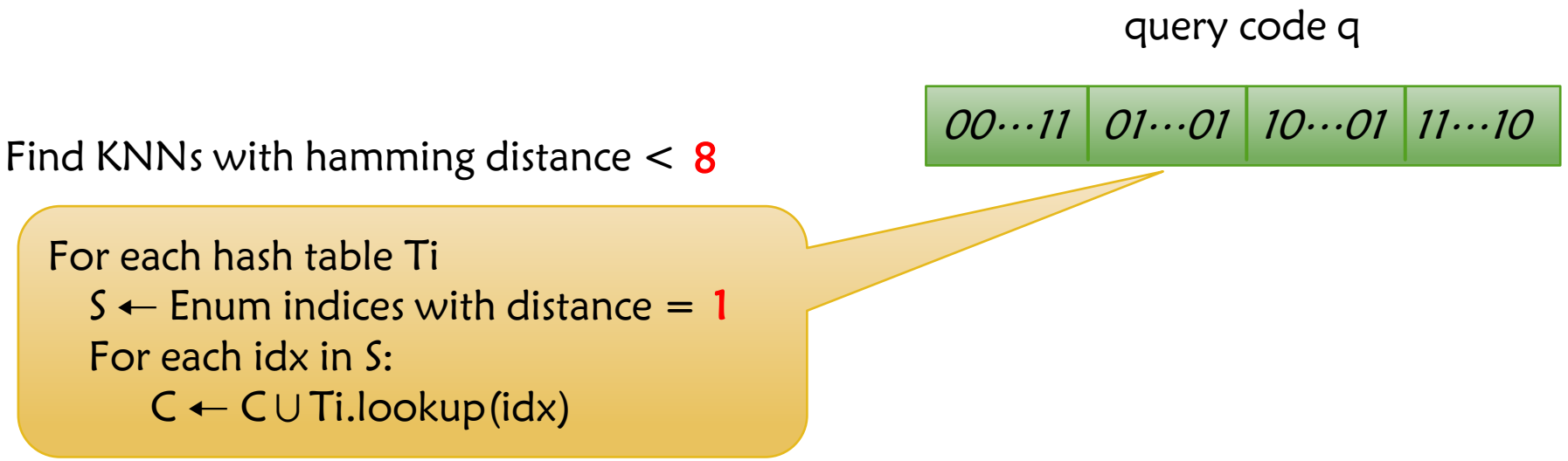
How to do KNN in binary space?

- To search 100 KNNs given a query code q



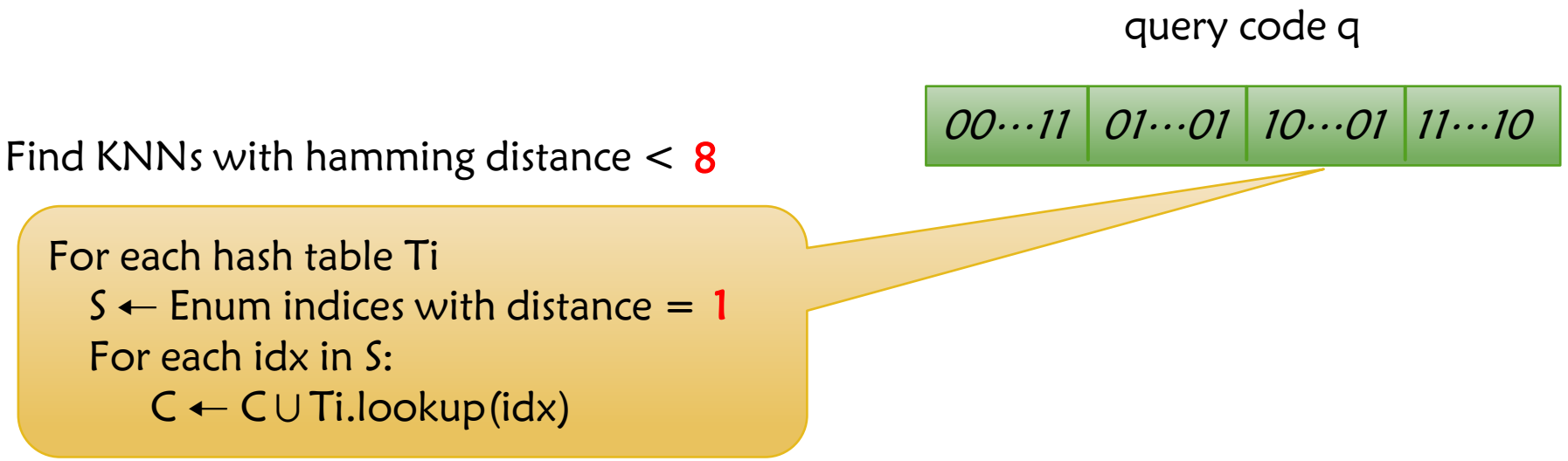
How to do KNN in binary space?

- To search 100 KNNs given a query code q



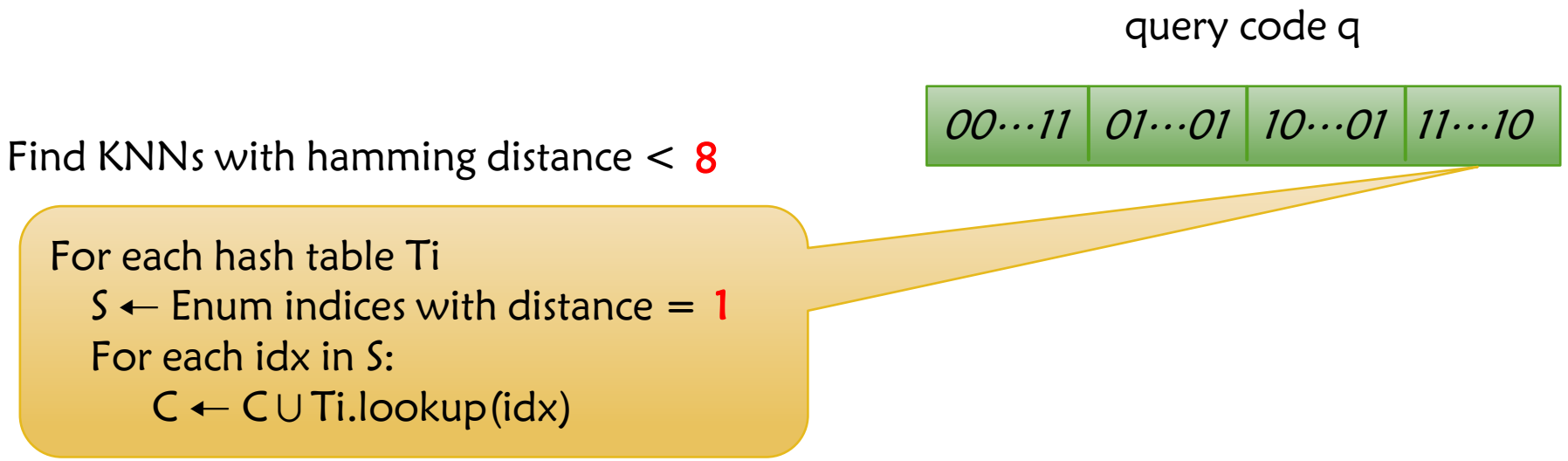
How to do KNN in binary space?

- To search 100 KNNs given a query code q



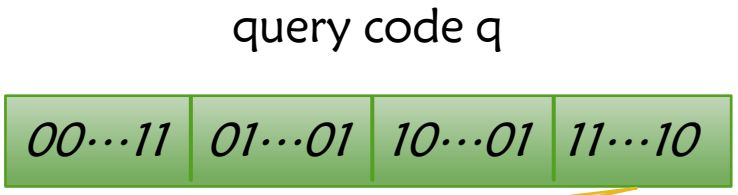
How to do KNN in binary space?

- To search 100 KNNs given a query code q



How to do KNN in binary space?

- To search 100 KNNs given a query code q



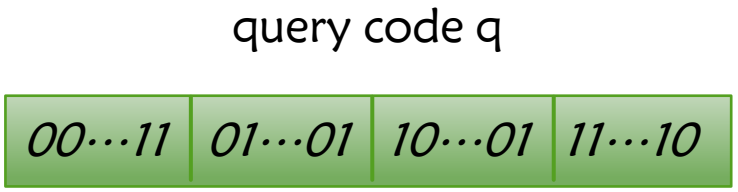
Find KNNs with hamming distance < 8

```
For each hash table Ti
    S ← Enum indices with distance = 1
    For each idx in S:
        C ← C ∪ Ti.lookup(idx)
```

```
For each image code x in C:
    if distance(x, q) < 8 :
        add x to result
if |result| >= 100:
    return KNN in result
```

How to do KNN in binary space?

- To search 100 KNNs given a query code q



For each d = 4, 8, 12, 16, ….

For each hash table Ti
  S ← Enum indices with distance = $\frac{d}{4} - 1$
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)

For each image code x in C:
  if distance(x, q) < **d** :
    add x to result
if |result| >= 100:
  return KNN in result

Optimization #1: approx. KNN

- To search 100 KNNs given a query code q

For each d = 4, 8, 12, 16, ….

For each hash table Ti
  S ← Enum indices with distance = $\frac{d}{4} - 1$
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)

For each image code x in C:
  if distance(x, q) < **d**:
    add x to result
if |result| >= 100:
  return KNN in result

Problem:
Large d → numerous (combinatorial) lookups

Typically, d=20 →
#lookups = 165K

Optimization #1: approx. KNN

- To search 100 KNNs given a query code q

For each d = 4, 8, 12, 16, ….

For each hash table Ti
  S ← Enum indices with distance = $\frac{d}{4} - 1$
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)

For each image code x in C:
  if distance(x, q) < **d**:
    add x to result
if |result| >= 100:
  return KNN in result

Problem:
Large d → numerous (combinatorial) lookups

Typically, d=20 →
#lookups = 165K

Our insight:

- Stop search as soon as the candidate set C is big enough
- KNNs in C approximates the true KNNs

Optimization #1: approx. KNN

- To search 100 KNNs given a query code q

For each d = 4, 8, 12, 16, ….

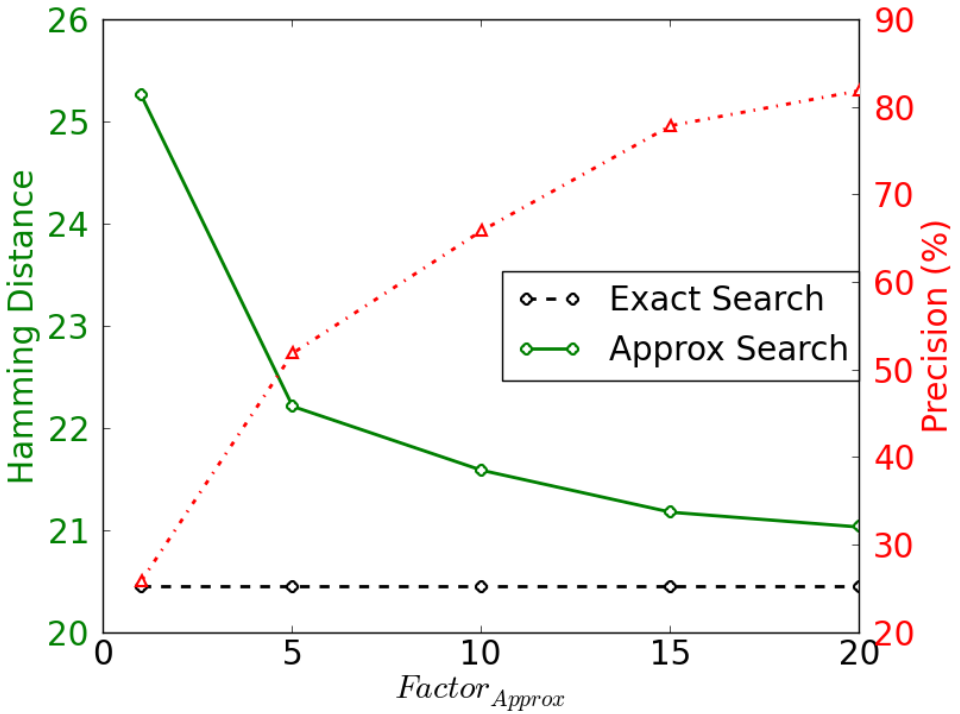
For each hash table Ti
  S ← Enum indices with distance = $\frac{d}{4} - 1$
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)
    if |C| >= f * 100:
      return KNN in result

Our insight:

- Stop search as soon as the candidate set C is big enough
- KNNs in C approximates the true KNNs

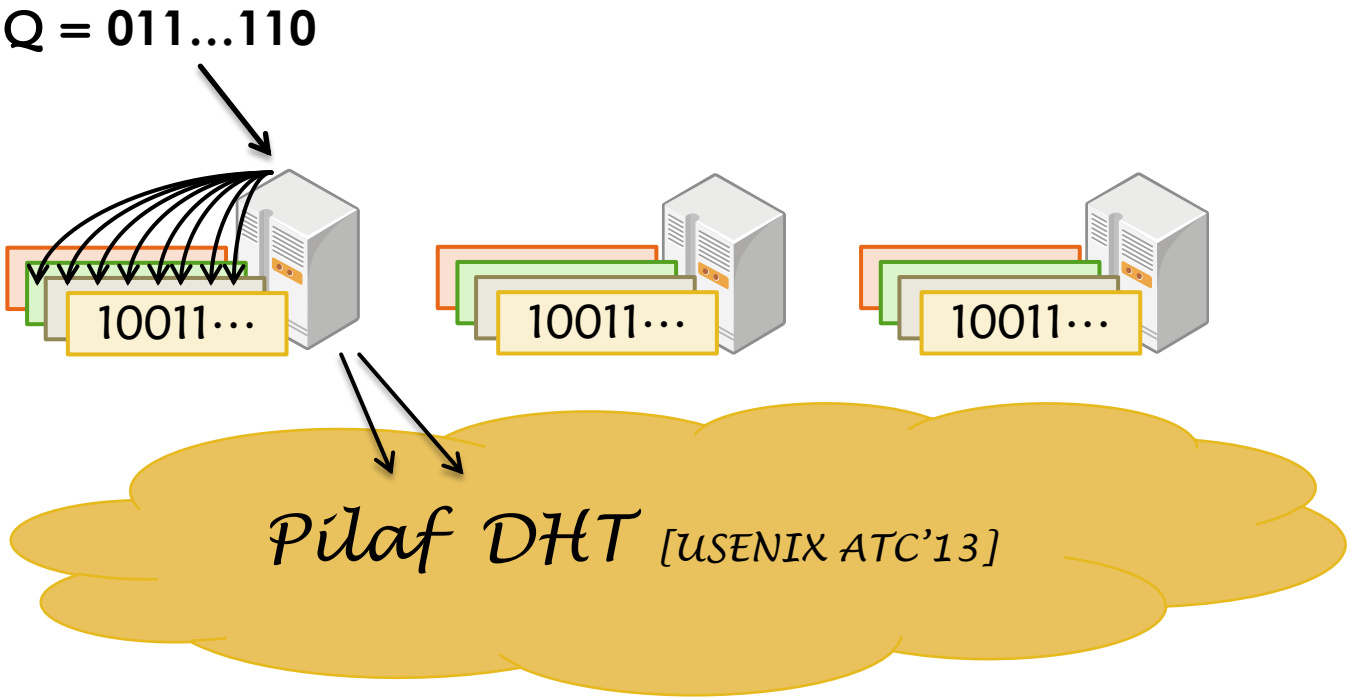
Optimization #1: approx. KNN

- Experiments show:
  - To obtain k results, we can stop search when $|C| > 20 * k$
    - Results contain 80% of true KNNs
    - Avg. distance of results is close to that of true KNNs (<1)
  - Reduces # of lookups by a factor of 40



Optimization #2: avoid null lookups

- Observation: >90% lookups return empty result
- Each search node keeps a bitmap for each hash table
  - Do a lookup in DHT only after the bitmap returns a hit
  - Bitmap size (4*500MB) does not increase with # of images indexed



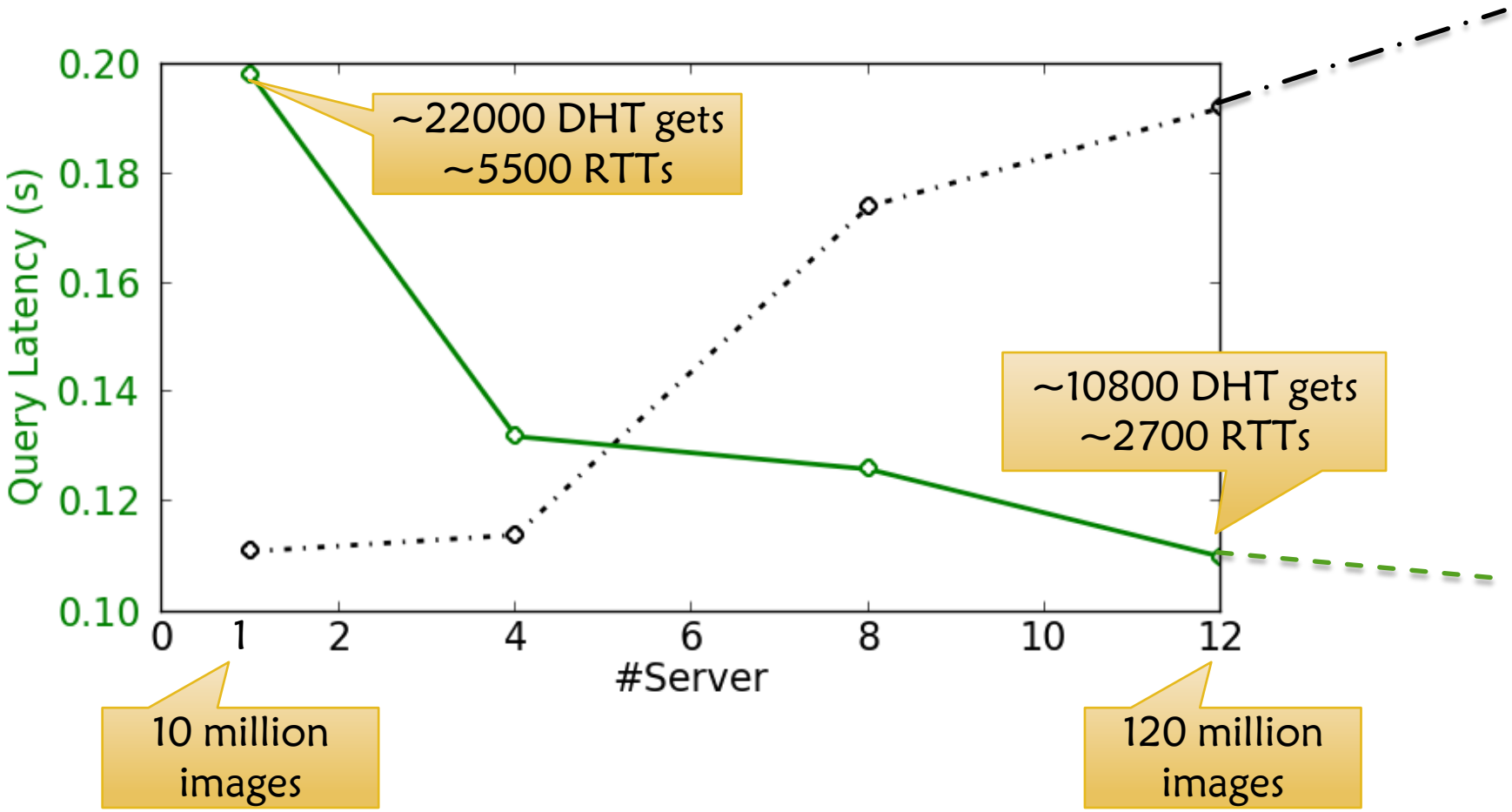
Outlines

1. Motivation
2. VertiCut Design
3. Evaluation
4. Related Work

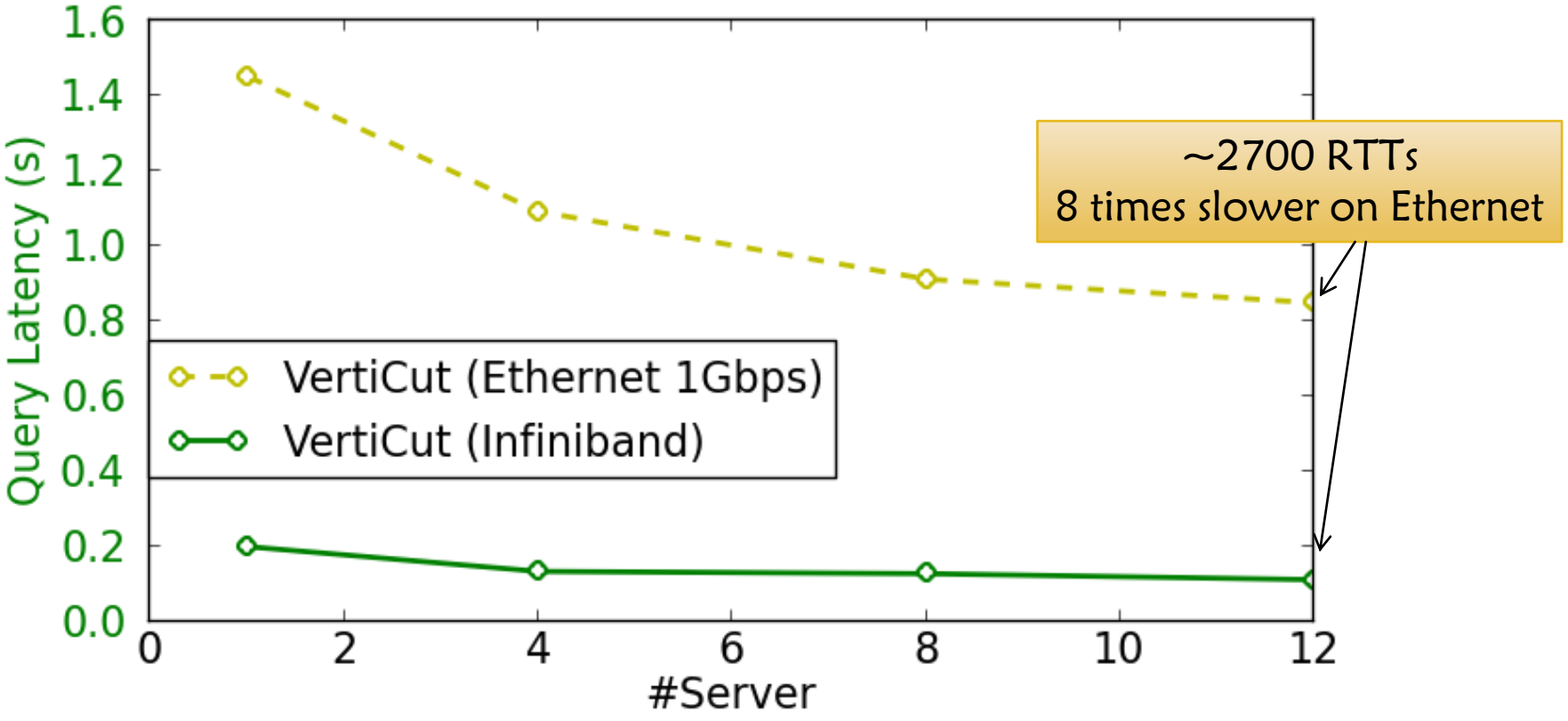
Experiment Environment

- Experimental Setup
  - 12 servers connected with 20Gbps Infiniband
  - 1 billion image descriptors from BIGANN dataset
  - Each query retrieves 1000 KNNs

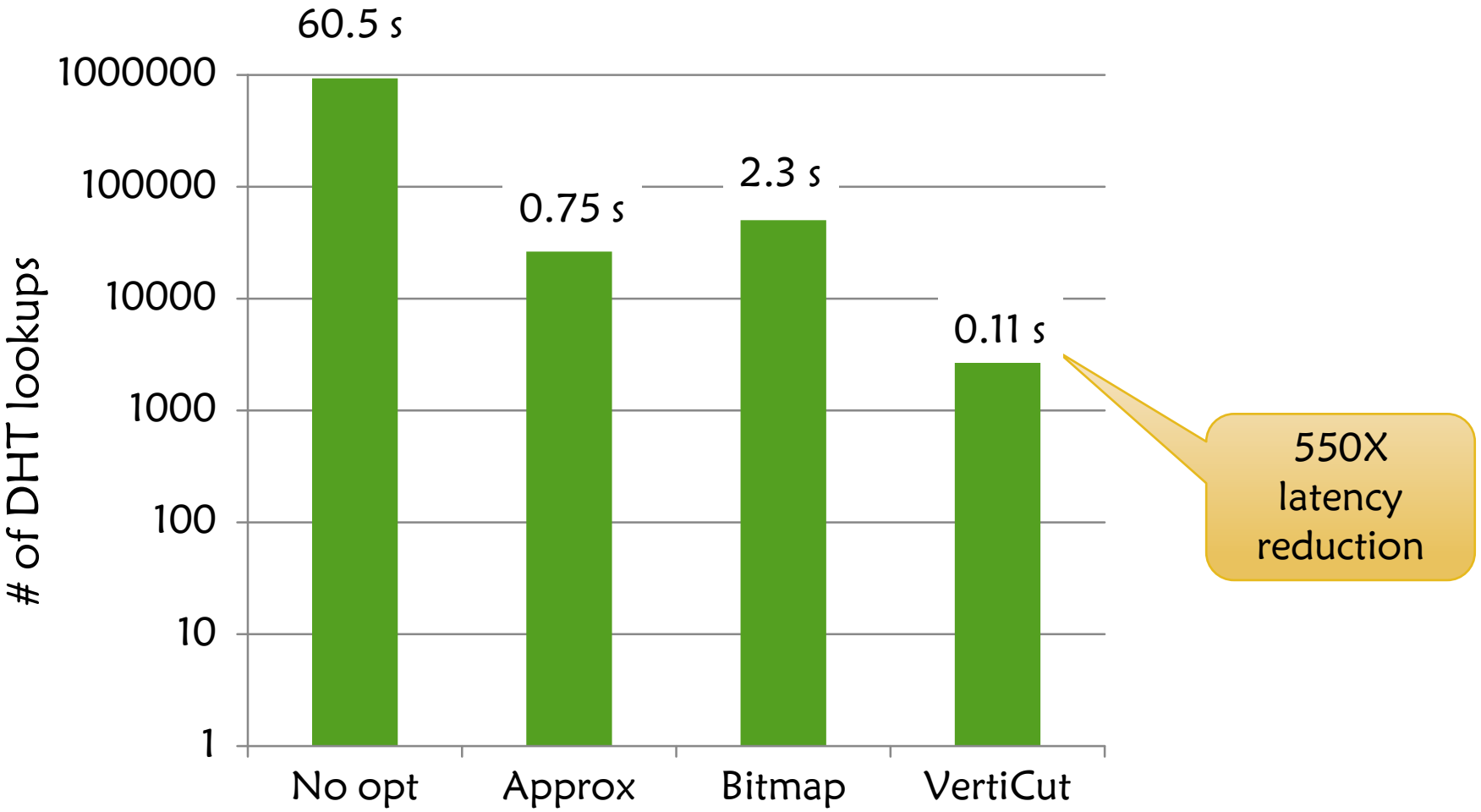
Vertical scales better than Horizontal



VertiCut is only feasible on low-latency network



Effects of Optimizations



Outlines

1. Motivation
2. VertiCut Design
3. Evaluation
4. **Related Work**

Related Work

- Bag-of-features based search
  - JI et al.[TM’13], MARÉE et al.[MIR’10], YAN et al.[SenSys’08], MIH[CVPR’12], Rankreduce[LSDS-IR’10]
  - Traditionally use horizontal partition for distribution
- High-dimentional search trees (e.g. KD-tree)
  - ALY et al.[BMVC’11]
  - Build a distributed tree offline → Cannot be incrementally updated

Conclusion

- Ultra low-latency networks allow vertical partition to perform better than traditional horizontal partition
- VertiCut: a scalable image search engine
  - Built on top of Pilaf DHT on Infiniband
  - Use two optimizations to reduce DHT lookups
    - Approximate nearest neighbor search
    - Eliminate empty lookups

Thank You!