Design, Implementation, and Evaluation of Scalable Content-Based Image Retrieval Techniques

Oral Examination

Presented by *Wyman Wong*

Supervised by *Prof. Michael R. Lyu*

June 15, 2007

Introduction

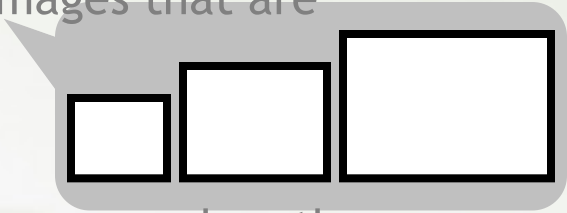
- Content-based Image Retrieval (CBIR)
  - The process of searching for digital images in large databases based on image contents
  - It consists of four modules in general:
    - data acquisition and processing
    - feature representation
    - data indexing
    - query and feedback processing

Introduction – Problem 1

- CBIR is extensively studied in both academia and industry
- Difficulties:
  - Large database, high-dimensional Image contents
- Previously Proposed Systems:
  - IBM's QBIC, Virage, MIT's Photobook, etc
  - Some work in small database only → *Exhaustive Linear Search*
  - Some employ PCA to reduce the dimension of feature → *Traditional multidimensional indexing techniques*
  - They suffer from **performance problem** in large database of high-dimensional features

Introduction – Problem 2

- To build a web-scale CBIR system
  - Fill the database using the WWW as a logical repository
  - Obtain web images by web image crawlers
- During web image crawling
  - Easy to encounter some web pages that contains images that are nearly the same
  - Near-duplicate images are likely to be returned to the user together
  - *Users will not be interested in seeing duplicated images* unless they are interested in that image
  - Moreover, it spams the image database
- Thus, web-scale CBIR systems suffer from **Duplication Problem**



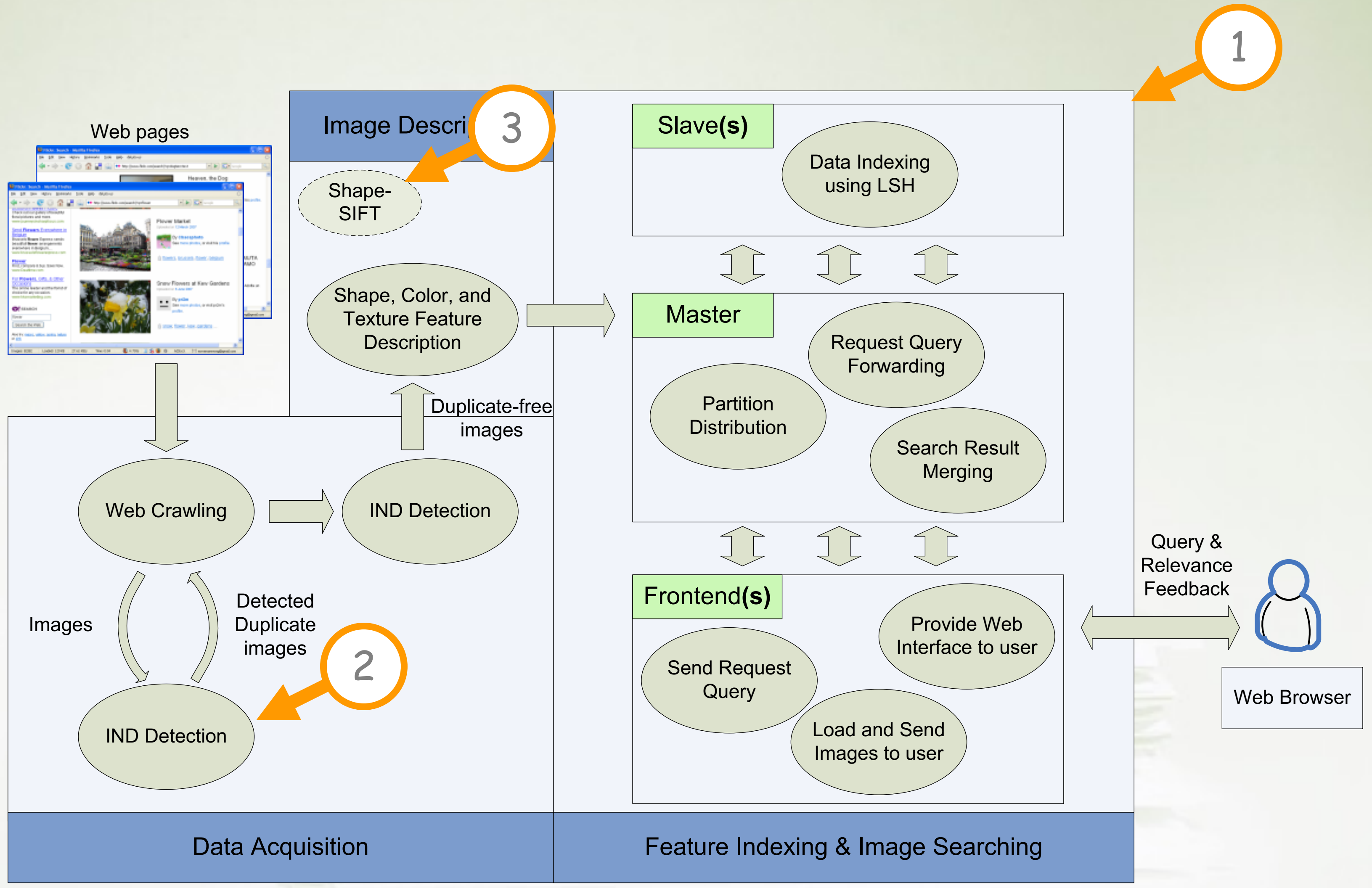
Introduction – Problem 3

- Traditional CBIR system employs global feature
- There are increasing interests in applying **local invariant feature** in CBIR systems
- **SIFT feature descriptor** is one of the state-of-the-art local invariant feature descriptor
- It performs the best in matching tasks and many research work employs this technique
- However, SIFT descriptor cannot be directly applied in CBIR because it is variant to **change in background and object color**
- This is the **invariance problem** of SIFT

Contribution

- Large-Scale CBIRS
  - **A novel scalable CBIR scheme** that employs locality-sensitive hashing (LSH)
  - **Comprehensive empirical performance evaluation** over one million images
- IND Detection System
  - **A novel IND detection scheme** that adopts SIFT and LSH
  - **A new match verification process and a new empirical distance metric**
- Local Descriptor for CBIR
  - **A new invariant local descriptor** that extends SIFT descriptor to achieve background and object color invariance

Overview of Contribution



A Distributed Scheme for Large-Scale CBIR

- Background & Related Work
  - Content-based Image Retrieval (CBIR)
    - **Similarity search** using image content
    - **Query by** providing an **example** image
    - Image contents can be **automatically derived** from the example
    - Similarity between images is defined by a **similarity measure**
    - **Fast indexing technique** is employed to speed up the search
    - **Relevance feedback** is used to progressively refine the search
      - Narrow down the **semantic gap** between high-level concept and low-level feature
      - Mark each search result as **relevant** or **irrelevant** to the query
      - Repeat the search with this additional information
    - **MIT's Photobook:** Suitable for small database (100 – 10,000) only
    - **IBM's QBIC, Lew's CBIRS, Egas's CBIRS:** PCA, *k*-d tree / R*-tree

A Distributed Scheme for Large-Scale CBIR

- Motivation
  - Traditional indexing techniques employed by CBIR can not scale up to a large database of high-dimensional features
  - Curse of dimensionality
  - Recently, LSH was proposed for solving near neighbor search problem in high dimensional spaces
    - M. Datar et al. reported that LSH searches for near neighbors 30 times faster than *k*-d tree does when dimensionality > 200
    - No prior dimension reduction is required
    - Accurate and fast
    - Large memory consumption

A Distributed Scheme for Large-Scale CBIR

- For CBIR, image contents are often represented in high dimensional spaces
- In our proposing CBIRS, image contents are represented by 238-element global feature vector
- Thus, LSH is very suitable for our system
- However, LSH has memory consuming problem
- We propose a parallel and distributed scheme to address the problem

Locality-Sensitive Hashing (LSH)

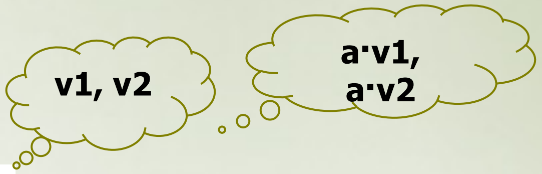
- An emerging new indexing algorithm
  - LSH *cannot* always find all the near neighbors
  - Instead, it ensures a near neighbor being found with a fixed *probability* in *sublinear* time
- Principles
  - Hash function is carefully designed such that

    the probability that two points share the same hash value decreases when the distance between them increases

  - By looking into the hash buckets of the query point, we obtain many near neigbhors of the query points
  - A large fraction of data points can be ignored

Locality-Sensitive Hashing (LSH)

- We employ the E²LSH (Exact Euclidean LSH)
- Hash function $h_{a,b}(v)$ is defined by:



$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor$$

  - *a: a d dimensional vector with entries chosen independently from a Gaussian distribution*
  - *b: a real number chosen uniformly from the range [0, w]*
- To locate a hash bucket of a query point **in a hash table**, *k* hash functions are used simultaneously
    - larger *k* reduces the chance of hitting data points that are not R-near neighbors
- There are ***L* sets of hash tables**
    - larger *L* increases the probability of finding all R-near neighbors
- The probability of finding near neighbors can be controlled by the parameters *L* and *k*

Main problem of *E^2LSH*

- *E^2LSH* is efficient in answering query
  - However, like any other memory-based LSH solutions, it has large memory consumption
  - Disk-based LSH → Slow in answering query
  - Memory-based LSH → Fast but …
- *E^2LSH* is a memory-based implementation
  - All the data points and the data structures are stored in the main memory
  - Maximum database size limited by the amount of free main memory available

Our Scalable Implementation

- We propose two multi-partition indexing approaches
  - Both divide the whole database into **multiple partitions**
  - Each partition corresponds to a **distinct portion** of the whole database and is associated with a *partition structure*
  - Each of them is **small enough** to be stored into the main memory
  - Then we can **process queries on each** of the partitions in memory
  - Two approaches
    - Disk-based multi-partition indexing
    - Parallel multi-partition indexing

Disk-Based Multi-Partition Indexing

- The algorithm
  - Employ a machine to
    1) load one partition structure at a time from disk into the main memory, and
    2) process query on it
  - These two steps are done sequentially and alternatively until the queries are processed on all partitions
  - The first step is very **time consuming**!

Problem with the Disk-Based Approach

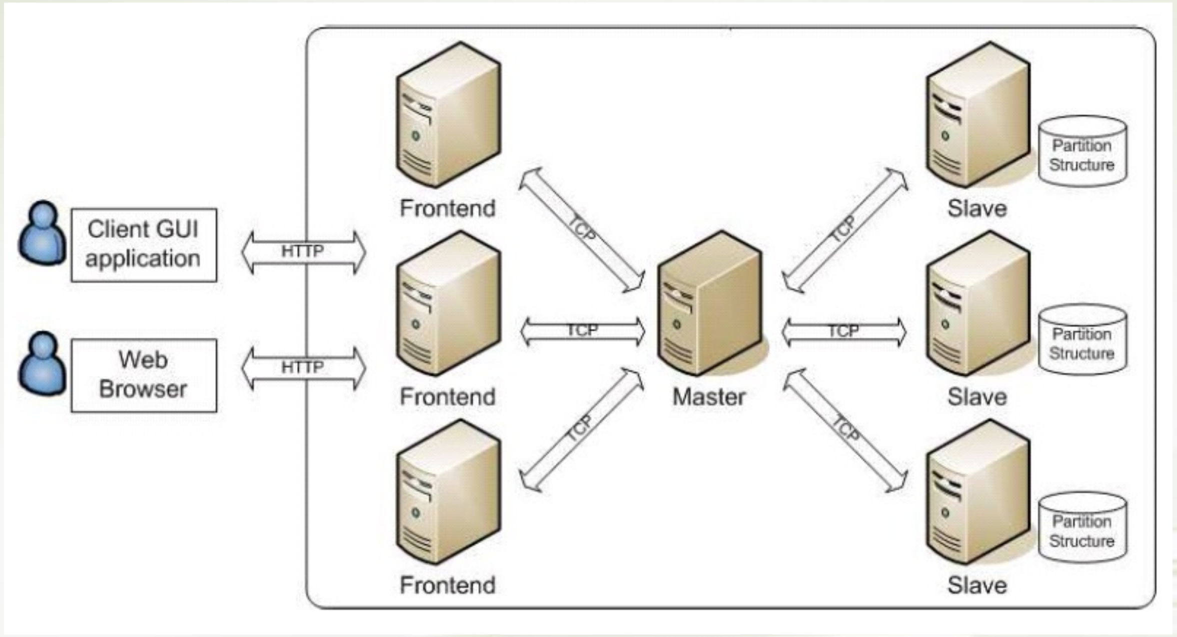
- **Disk-access overhead** for loading the hash tables into the main memory
- All partitions of a 0.5 million database → **750 sec**
- **Not** suitable for **real-time application**

- Thus, we propose a parallel and distributed solution to overcome this overhead issue and speedup the overall solution

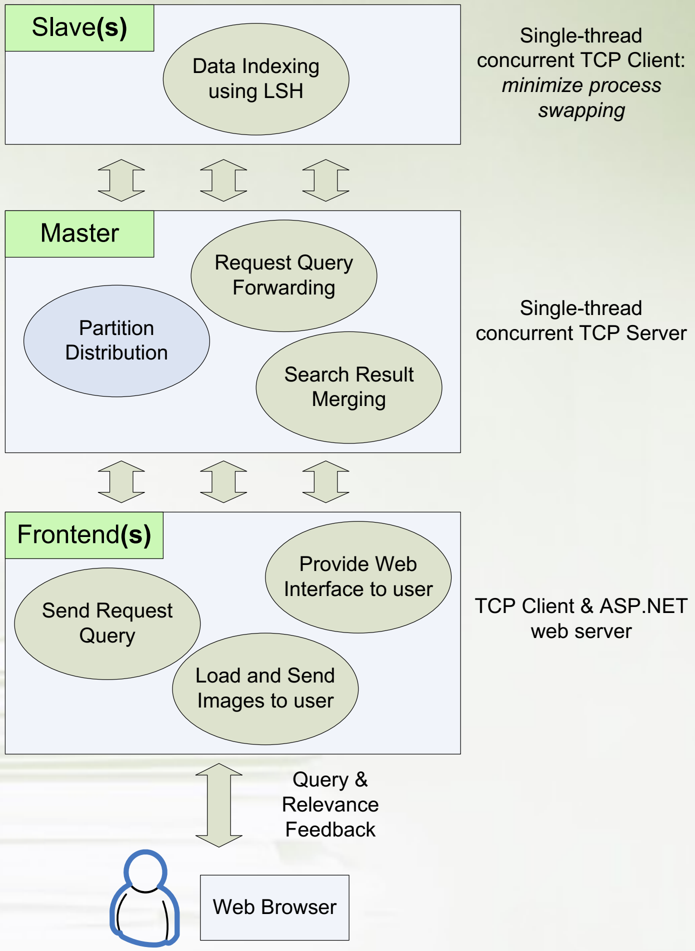
Parallel Multi-Partition Indexing

- The algorithm
  - Distribute the data indexing tasks over **a cluster of *Slaves***
  - Each Slave answers queries over **one partition** of the database
  - Slaves store the assigned partition structures in main memory **permanently**
  - **No re-loading** of the partition structures
  - **No disk-access overhead**
  - Slaves are managed by **Master**
  - Master listens to the query requests from **Frontend** and then forward the requests to Slaves
  - Query answers from Slaves are merged in Master and returned to Frontend

System Architecture



System Diagram



Slave(s)
Data Indexing using LSH
Single-thread concurrent TCP Client: *minimize process swapping*
Master
Request Query Forwarding
Partition Distribution
Search Result Merging
Single-thread concurrent TCP Server
Frontend(s)
Provide Web Interface to user
Send Request Query
Load and Send Images to user
TCP Client & ASP.NET web server
Query & Relevance Feedback
Web Browser

System Speed

- Network transmission delays exist in every communication
- To minimize the adverse effect of the network delay, we maximize the efficiency of communication
  - Image is represented by a 4-byte integer image ID number in query and reply messages
  - Query message and reply message are represented by a *Structure* data structure in C++
    - Large Integer and floating-point number inside the structure are represented in **Binary format**, which is both accurate and space-saving
    - No parsing of value is required
- Query message
  - < 1500 bytes, can store a 238-dimensional query feature and 120 +ve / -ve relevance feedback image ID numbers
- Reply message
  - < 1500 bytes, can store up to top 150 ranking image ID numbers
- Total system's query processing time

$$T_{Total} = T_{Frontend} + RTT + T_{Master} + RTT + MAX(T_{Slave_1}, T_{Slave_2}, \ldots, T_{Slave_N}$$

Disk-Loading for Relevance Feedback

- We allow users to do **relevance feedback (RF)**
- Thus, the query message must contain **relevance feedback info.**
- Slaves need to obtain the **feature vectors** of the RF images
- Solution
  - Fetch the feature vectors from the disks upon request
  - # feature vectors needed in each query is at most 120 which is not many
- Advantages
  - No need to consume the precious memory space
  - No need to load all data points into the main memory (not scalable!)
  - No need to transmit feature vectors between machines
- The solution can be accelerated by two tricks
  - Save the database of feature vectors in Binary format
  - Sort the relevance feedback image ID and query according to sorted order
    - Hard disk head can read all the vectors by moving in a single direction (shorten disk seek time!)

Advantages of the Parallel Approach

- No disk-access overhead
  - Slaves store partitions permanently in memory
  - No disk-access overhead
- Reduced retrieval time by parallelization
  - System's retrieval time is reduced by a multiple equals to the number of employed Slaves
  - The tradeoff is hardware resource
- Guaranteed retrieval time
  - System's retrieval time depends mostly on the sizes of partitions in the Slaves
  - By maintaining the size of partition for each Slave below the maximum size, we guarantee the retrieval time to be below a small fixed value

Feature Representation

- We represent an image with three types of visual feature: color, shape, and texture
- For color, we use Grid Color Moment
- For shape, we employ an edge direction histogram
- For texture, we adopt Gabor feature
- In total, a 238-dimensional feature vector is employed to represent each image in our image database

Empirical Evaluation

- Experimental Testbed
  - Testbed containing 1,000,000 images crawled from Web
  - 5,000 images from COREL image data set are engaged as the query set
    - Contains 50 categories
    - Each category consists of exactly 100 images that are randomly selected
    - Every category represents a different semantic topic, such as butterfly, car, cat, dog, etc

Performance Metrics

- Two standard performance metrics
  - Precision and recall

$$Recall = \frac{\#\ relevant\ images\ in\ the\ returned\ images}{total\ \#\ relevant\ images\ in\ the\ database}$$

$$Precision = \frac{\#\ relevant\ images\ in\ the\ returned\ images}{total\ \#\ returned\ images}$$

  - Relevant image if it is one of the 100 COREL images in the database which share the same category with the query images
  - Evaluate efficiency by average CPU time elapsed on a given query

Experimental Setup

- Form a query set of 1000 image examples by randomly select 20 images from each of the 50 COREL image categories
- A database of size *N* contains the followings:
  - 5,000 COREL images
  - *N*-5000 other images selected from our testbed
- Extract the image features using the techniques discussed

Experimental Setup

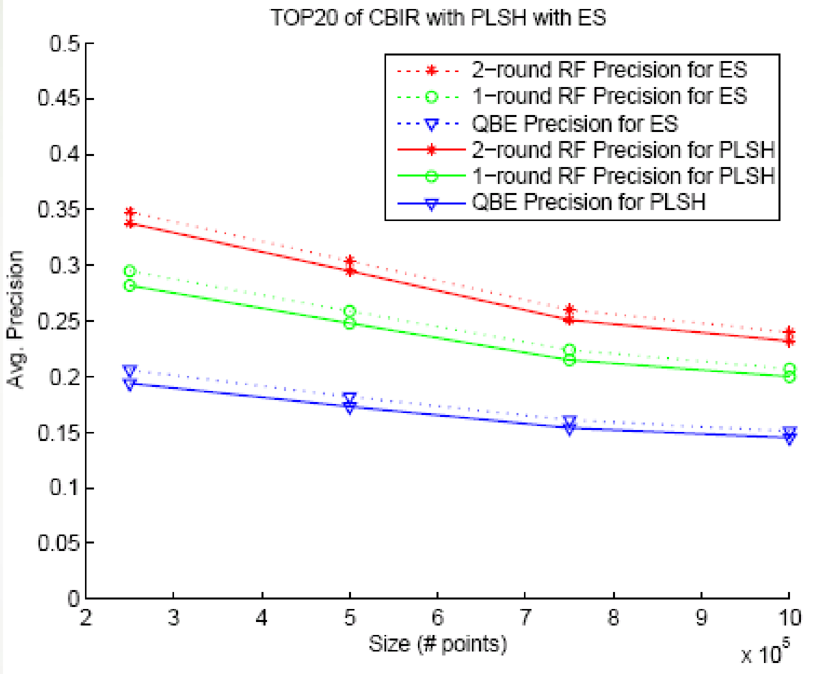
- Perform a series of experiments using our CBIR system with LSH indexing on all databases
  - LSH’s parameters: L = 550 and k = 34
  - Retrieve top 20, 50, and 100 ranking images for each query image
  - Calculate recall and precision
- Simulate two rounds of relevance feedback
  - Re-querying the database with relevant examples with respect to the given query
- Environments
  - Slaves (C++)
    - 3.2GHz Intel Pentium 4 PC with 2GB memory running Linux kernel 2.6
  - Master (C++)
    - Sun Blade 2500 (2 x 1.6GHz US-IIIi) machine with 2GB memory running Solaris 8
  - Frontend (C#)
    - Nix dual Intel Xeon 2.2GHz with 1 GB memory running Linux kernel 2.6 (Evaluation)
    - ASP.NET server on a 3.2GHz PC with 1GB memory running Windows OS (Deployment)
- The same experiments are repeated with Exhaustive Linear Search (ES)

Experimental Results

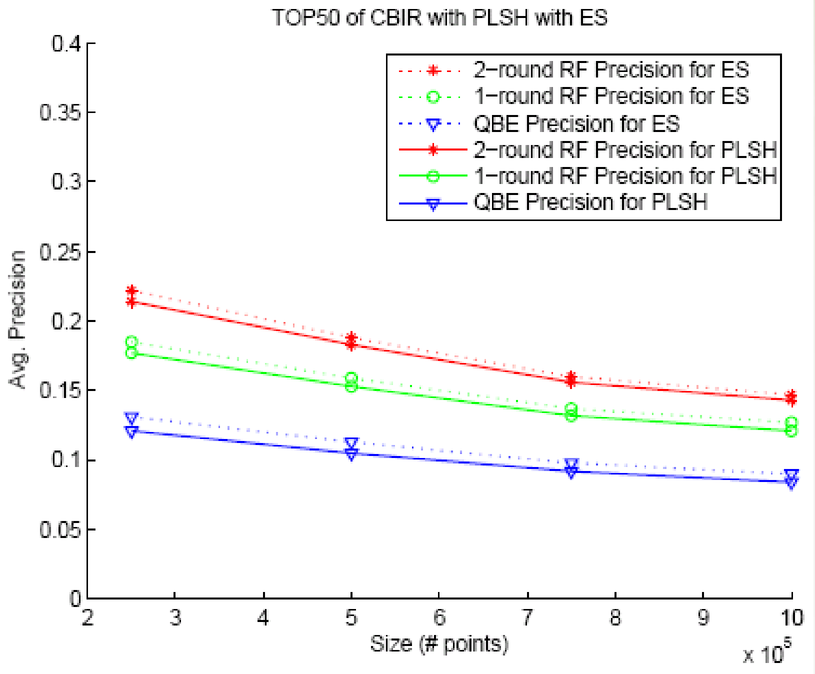
- Parallel-based multi-partition indexing approach (PLSH)
  - System performance is related to the number of Slaves we used
  - We present the performance of the **parallel system that uses the minimum possible number of Slaves**
    - **Minimize the demand on the precious hardware resource**
    - **Utilize the usage of the currently available hardware resource**
  - Maximum partition size of Slave is **0.25 million** data points

Average Precision of TOP20 & TOP 50

- The results of LSH is **very close to the ES** results
- Their maximal difference is **no more than 5%** at any database size



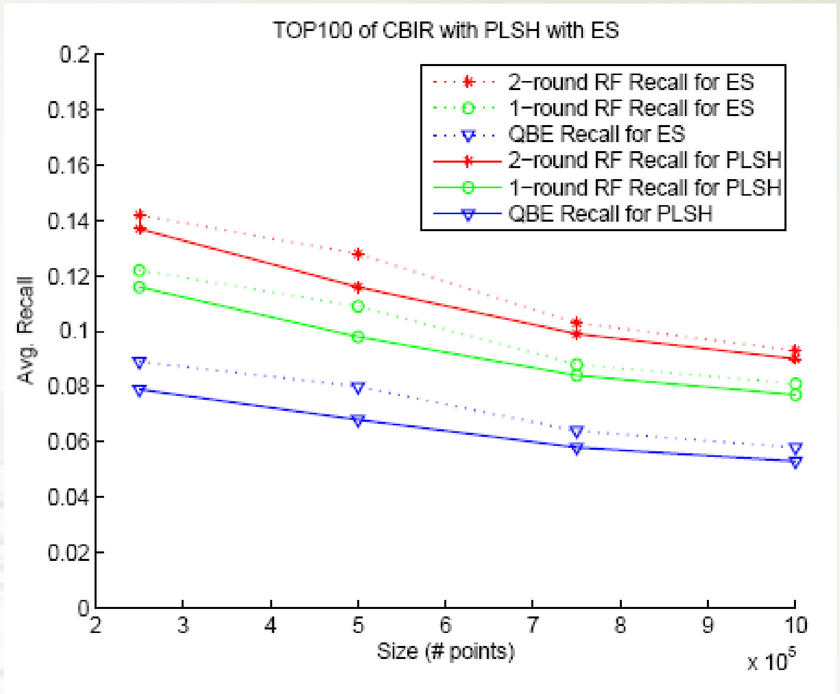
(a) Avg. Precision of TOP 20



(b) Avg. Precision of TOP 50

Average Recall of TOP100

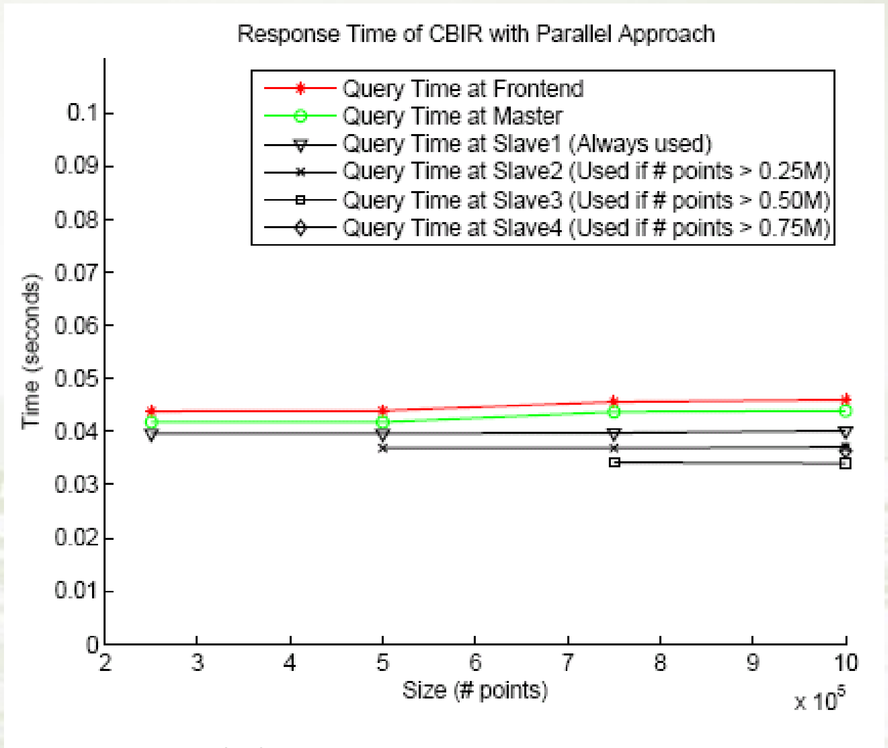
- Average recall and average precision of our CBIR system decrease with the database size, yet the **decreasing rate diminishes** when the database size increases
- **Relevance feedback** implemented in our system significantly improves the recall and precision



(c) Avg. Recall of TOP 100

Average Query Time

- The system query processing time is nearly constant for any size of database



(d) Avg. Query Time

Query Time of LSH over ES on different databases

- Taking the disk access overhead into account, **DLSH system is not faster than the ES system**
- **PLSH is a a lot faster than the ES system**
- The last column purely illustrates **the advantage of using LSH over ES**

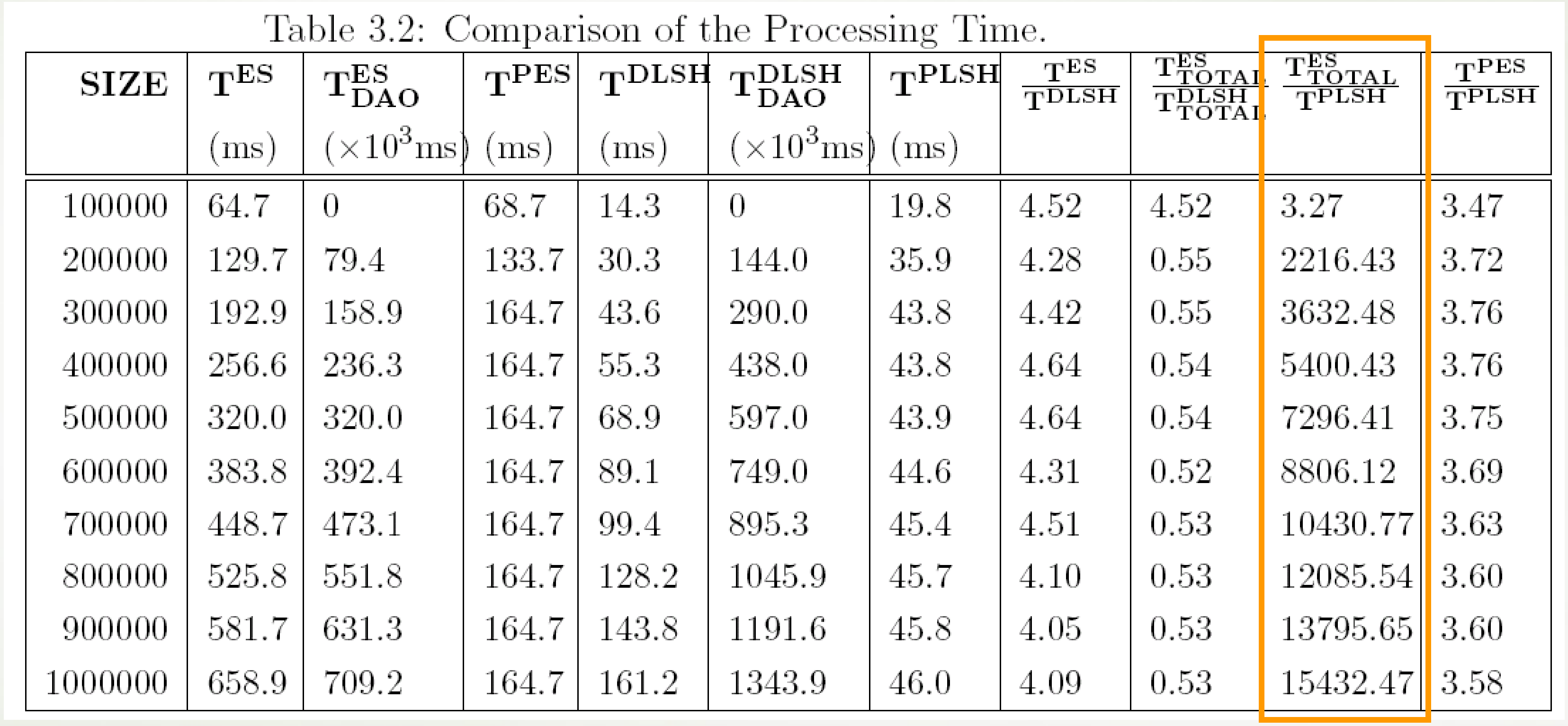
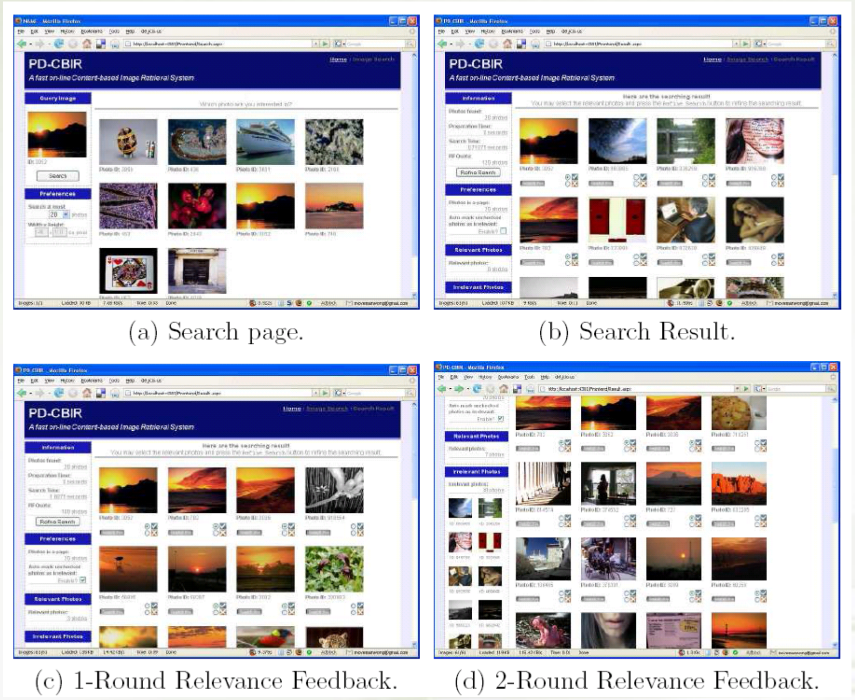


Table 3.2: Comparison of the Processing Time.

SIZE	T^{ES} (ms)	T^{ES}_{DAO} ($\times 10^3$ms)	T^{PES} (ms)	T^{DLSH} (ms)	T^{DLSH}_{DAO} ($\times 10^3$ms)	T^{PLSH} (ms)	$\frac{T^{ES}}{T^{DLSH}}$	$\frac{T^{ES}_{TOTAL}}{T^{DLSH}_{TOTAL}}$	$\frac{T^{ES}_{TOTAL}}{T^{PLSH}}$	$\frac{T^{PES}}{T^{PLSH}}$
100000	64.7	0	68.7	14.3	0	19.8	4.52	4.52	3.27	3.47
200000	129.7	79.4	133.7	30.3	144.0	35.9	4.28	0.55	2216.43	3.72
300000	192.9	158.9	164.7	43.6	290.0	43.8	4.42	0.55	3632.48	3.76
400000	256.6	236.3	164.7	55.3	438.0	43.8	4.64	0.54	5400.43	3.76
500000	320.0	320.0	164.7	68.9	597.0	43.9	4.64	0.54	7296.41	3.75
600000	383.8	392.4	164.7	89.1	749.0	44.6	4.31	0.52	8806.12	3.69
700000	448.7	473.1	164.7	99.4	895.3	45.4	4.51	0.53	10430.77	3.63
800000	525.8	551.8	164.7	128.2	1045.9	45.7	4.10	0.53	12085.54	3.60
900000	581.7	631.3	164.7	143.8	1191.6	45.8	4.05	0.53	13795.65	3.60
1000000	658.9	709.2	164.7	161.2	1343.9	46.0	4.09	0.53	15432.47	3.58

Application



(a) Search page. (b) Search Result.

(c) 1-Round Relevance Feedback. (d) 2-Round Relevance Feedback.

- We have built a **web-based ASP.NET Frontend** to demonstrate the real-time performance of the PLSH system
- Here are some screenshots of **sunset** search using our web-based Frontend application, **PD-CBIR**, over 1 million image database

Summary

- Proposed a scalable CBIR scheme based on a fast high dimensional indexing technique, LSH
- Conducted extensive empirical evaluations on a large testbed of a half million images
- Suggested a parallel and distributed solution over the shortcoming of LSH

Future Work

- Improve the system by minimizing the usage of Slaves
- Currently, for each search query, the system **gets all the Slaves involved**, no matter whether a similar image can be found in each Slave
- If we know there is no data point within a threshold radius from the query point in certain Slave, we need not to perform similarity search in that Slave
- Eventually, the system can process more search queries simultaneously
- We need a new dispatcher
  - We suggest to use PCA to reduce the dimension of the query point,
  - And apply *k*-d tree to determine at which Slave similar data points of query point would be located

Image Near-Duplicate Detection

Introduction

- Removing duplicated images prior to the return of results is critical in improving the quality of CBIR search
- We consider using invariant local features
- Invariant local features has been applied in many CV applications
  - Panorama Stitching
  - Image Retrieval
  - Object Recognition

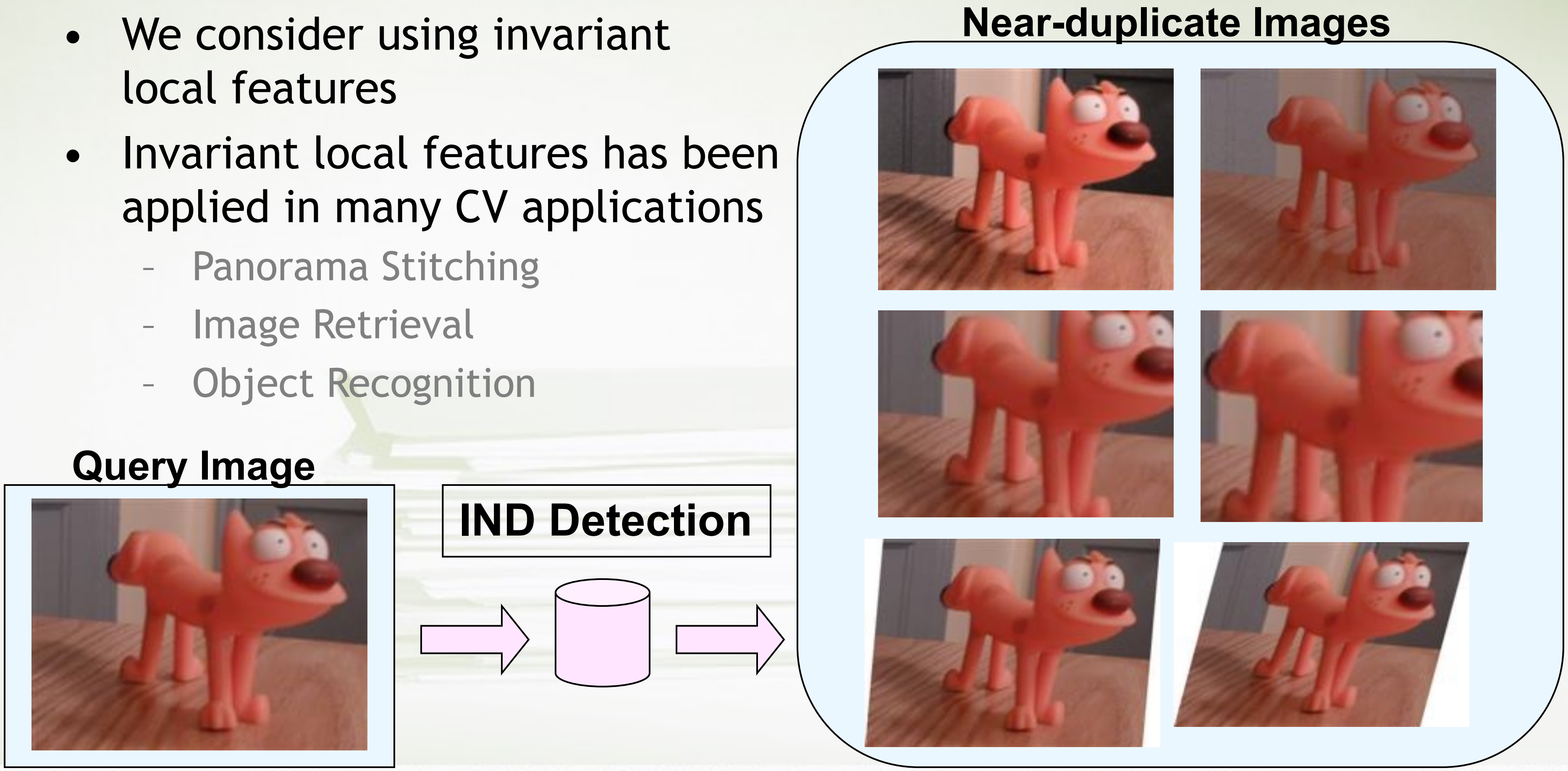


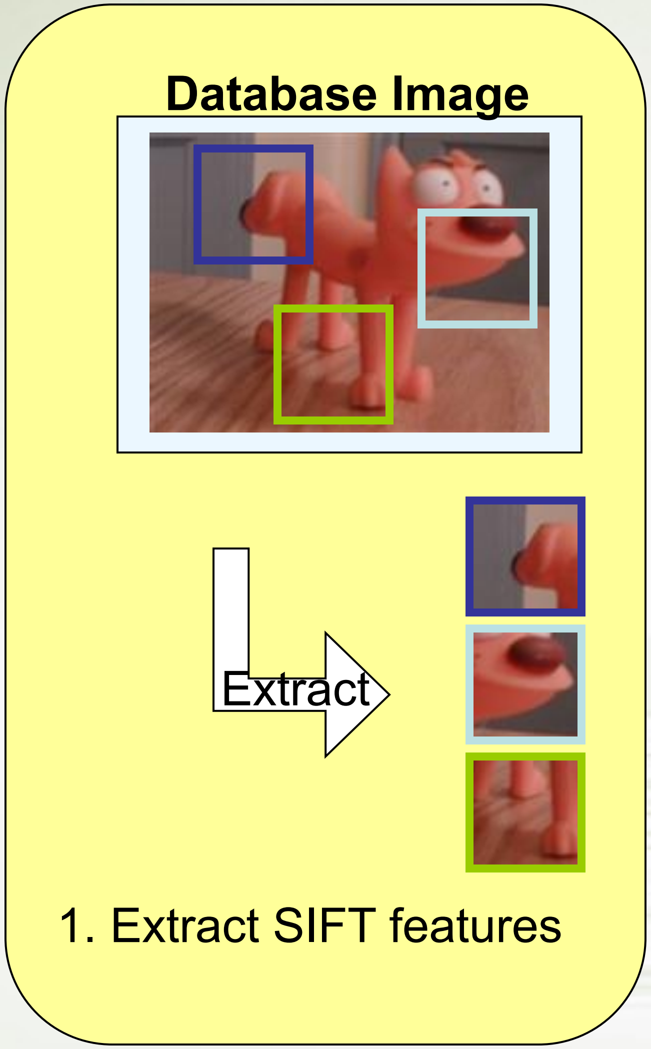
Image Near-Duplicate (IND) Detection

- Image near-duplicate (IND) detection
  - To remove meaningless duplicates in the returned result of content-based image retrieval system (CBIRS)
  - *To detect illegal copies of copyright images on the Web*
- Integrate a number of state-of-the-art techniques
  - **Detector**: DOG pyramid from SIFT
  - **Descriptor**: SIFT descriptor
  - **Matching**: Exact Euclidean Locality Sensitive Hashing (E^2LSH)
- Propose two new steps to improve recall & precision
  - A new matching strategy: *K-NNRatio matching strategy*
  - A new verification process: *Orientation verification process*

Two Main Phases

- Database Construction
  - Image Representation
  - Index Construction
  - Keypoint and Image Lookup Tables Creation
- Database Query
  - Matching
  - Verification Processes
  - Image Voting

Database Construction – Image Representation



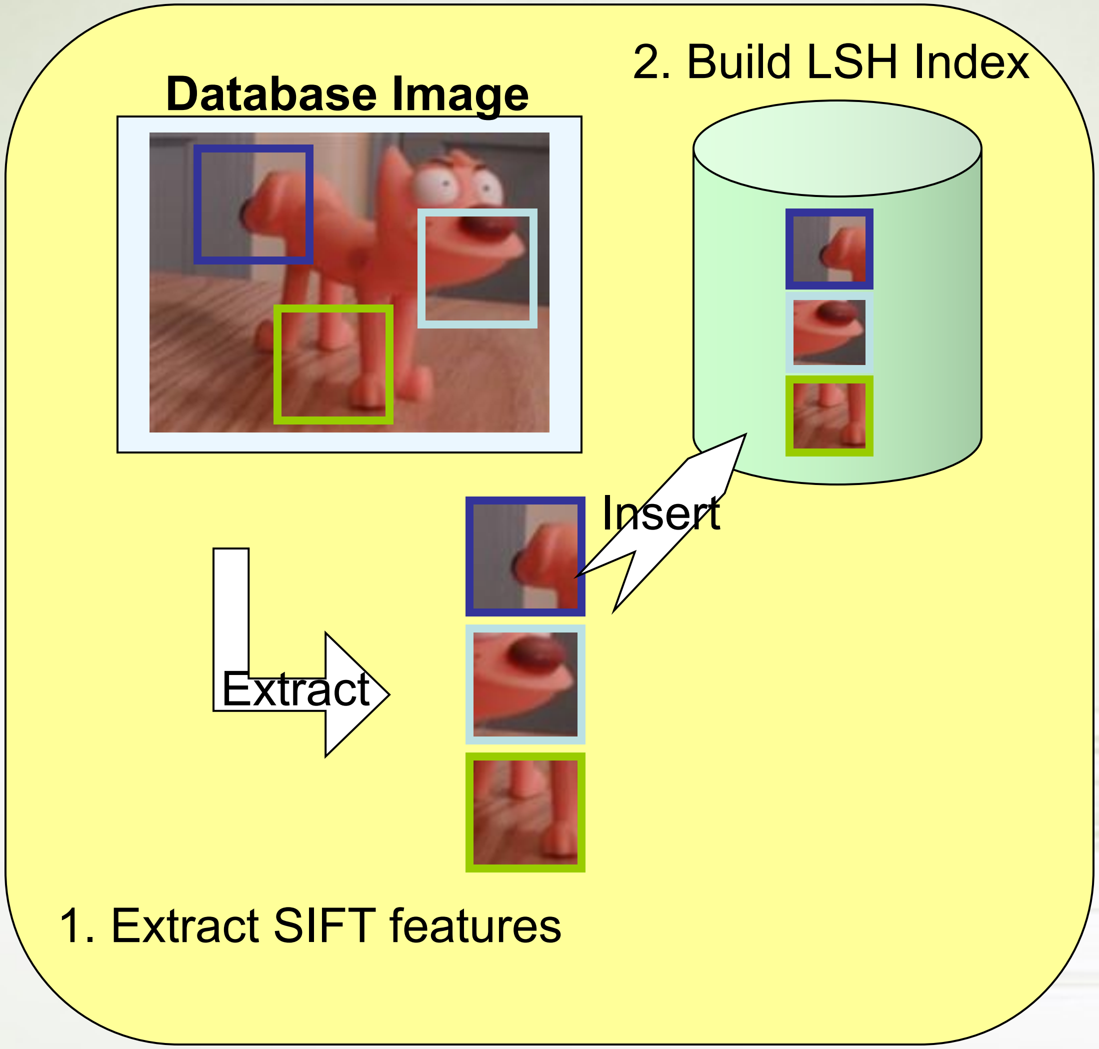
1. Extract SIFT features

- SIFT features are invariant to common image manipulations which include changes in:
  - *Illumination*
  - *Contrast*
  - *Coloring*
  - *Saturation*
  - *Resizing*
  - *Cropping*
  - *Framing*
  - *Affine warping*
  - *Jpeg-compression*

Huge amount of SIFT features in the database

Slow!

Database Construction – Index Construction



- Build LSH index for fast features retrieval
- E^{2}LSH allows us to find the *k*-nearest neighbors (*k*-NN) of a query point at certain probability

Database Construction – Index Construction

- E^{2}LSH's scalability problem
  - All data points and data structures are stored in the main memory
  - **Limitation:** Database's size < Amt. of free M.M.

- Solutions:
  - Offline Query Applications:
    - Adopt the disk-based solution that we presented eariler
  - Online Query Applications:
    - Adopt the parallel and distributed solution that we presented earlier

Database Construction - Keypoint and Image Lookup Tables Creation

- To reduce the main memory usage, we do not store the **details of the keypoints** and their **corresponding images** in the hash table
- We store them in **separate files on disk**
- To facilitate fast disk memory access
  - The details of a keypoint are stored in **a line** in keypoint lookup table

Keypoint lookup table schema

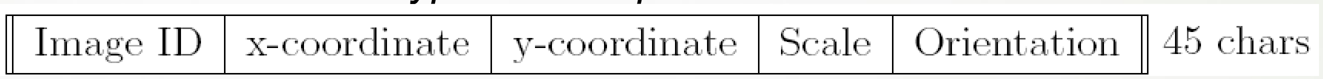


Image ID	x-coordinate	y-coordinate	Scale	Orientation	45 chars

  - The details of an image are stored in **a line** in image lookup table

Image lookup table schema

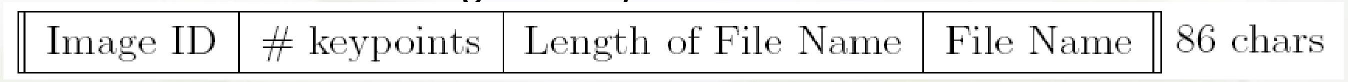
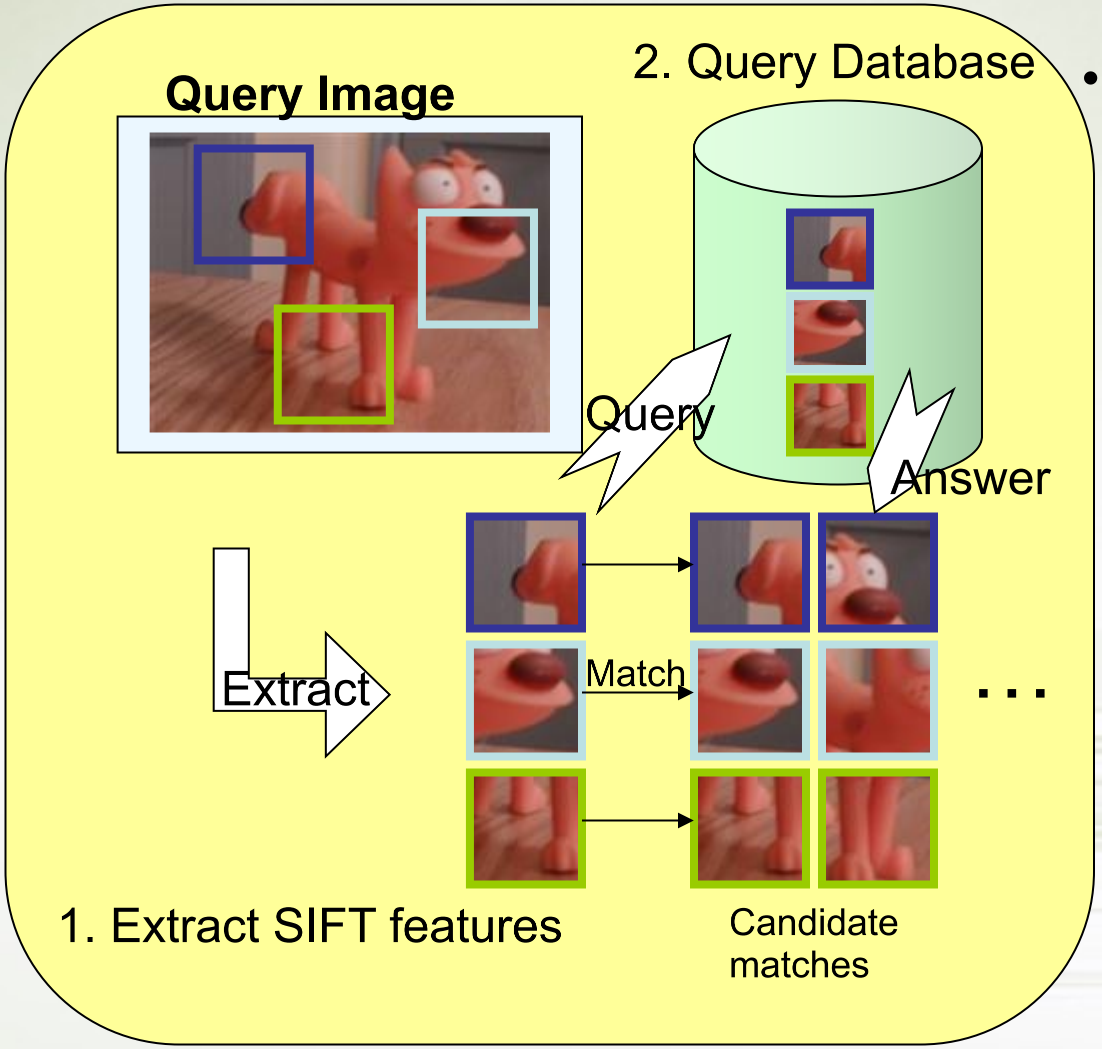


Image ID	# keypoints	Length of File Name	File Name	86 chars

Database Query



- Matching Strategies:
  - *Threshold Based*
    - Set threshold (**fixed**) on distance
  - *k-NN*
    - Set threshold (**fixed**) on distance
    - Take top *k* (**fixed**) nearest neighbors
  - *k-NNRatio*
    - Assign weights to *k* nearest neighbors
    - Depend on distance ratio and place

More Flexible!

Database Query – 1. Matching

- *k*-NNRatio
  - Weights are assigned based on two principles:
    - If a keypoint is nearer to the query point, it is probably the correct match and thus its weight should be higher
    - If a keypoint has large distance ratio to its next nearest neighbor, it is more likely to be the correct match and thus its weight should be higher
  - Weight of a keypoint is formulated as follow:

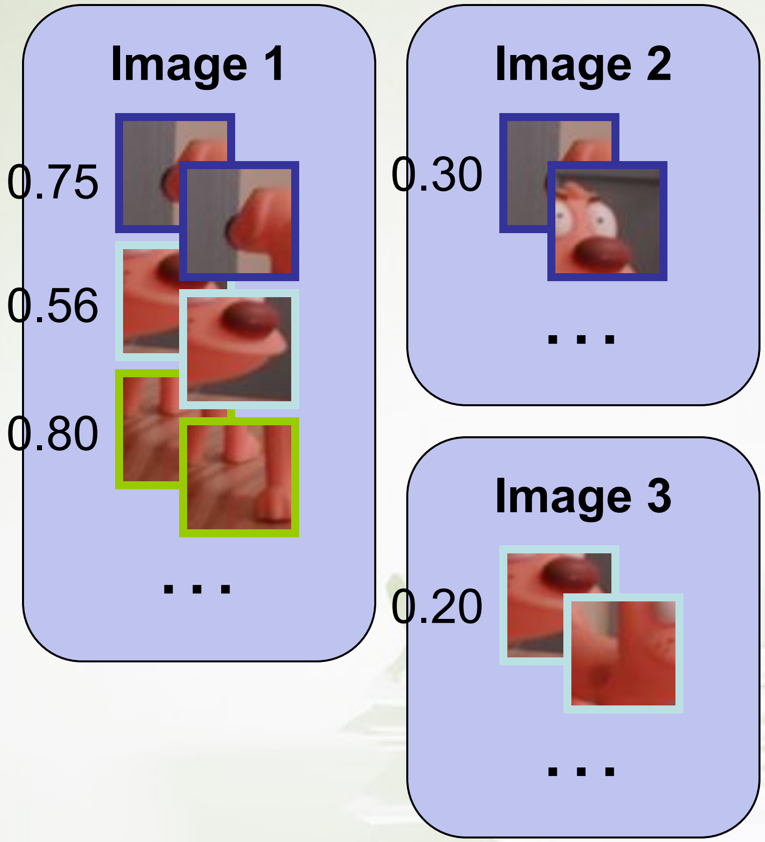
$$Weight(K_A) = \left(\frac{a}{k(K_A)}\right)^b \times \left(\frac{dist(K_B, Q)}{dist(K_A, Q)}\right)^c$$

    - *k*(K_A): the **place** of the point K_A in the top *k* ranking images
    - dist(K_A, Q): the **distance** between point K_A and Query point Q
    - Parameter b and c control the importance of the two terms
    - **b = c = 0** is a **special case of *k*-NNRatio** in which *k*-NNRatio matching strategy is reduced to *k*-NN matching strategy

Database Query – 1. Matching

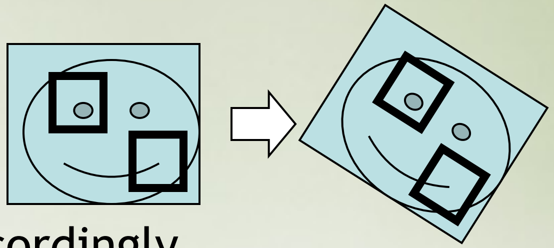
- Some advantages of *k*-NNRatio over *k*-NN
  - Nearest neighbor does not always gain high weight
    E.g. Weight will not be high if it is far from the query point
  - Keypoint with larger place (i.e. $k(K_A)$) can still gain high weight if it is far from the remaining neighbors
  - The *k* nearest neighbors do not get the same weight. If there are only two possible matches, ideally only two keypoints will get high weights. This relieves the adverse effect of the fixed value of *k* and the threshold.

Database Query – 2. Verification Processes

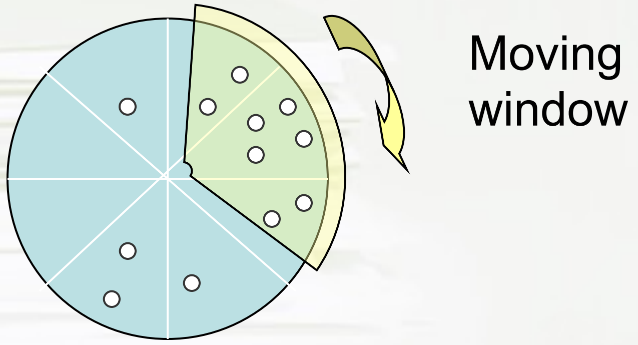


- Each query keypoint matches a number of keypoints in the database
- Each matched keypoint in the database corresponds to one image
- Group the matched keypoints by their corresponding images (ID)
- False matches can be discovered by checking the inter-relationship between the keypoints in the same group
  - However, not all false matches can be removed
- Others suggested affine geometric verification using RANSAC
- We further propose **orientation verification**

Database Query – Orientation Verification



- For each group
  - Observation:
    - If an image is rotated, all keypoints rotate accordingly
    - The changes in orientation of all keypoints are more or less the same
  - For each candidate match, find the **difference of the canonical orientation** between the query keypoint and the matched database keypoint
  - **Map** the difference to one of the 36 bins that cover 360 degrees
  - **Slide** a moving window of width 3 bins over the 36 bins and find a window that contains the **maximum number of supports**
  - **Remove** candidate matches in all the uncovered bins



Simplified Version: 8 bins

Database Query – Affine Geometric Verification

- The affine transformation between two images can be modeled by the following equation:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \\ 1 \end{bmatrix}$$

x: the homogenous coordinates of a keypoint *in the query image*
b: the homogenous coordinates of the matched keypoint *from the database*
A: the transformation matrix

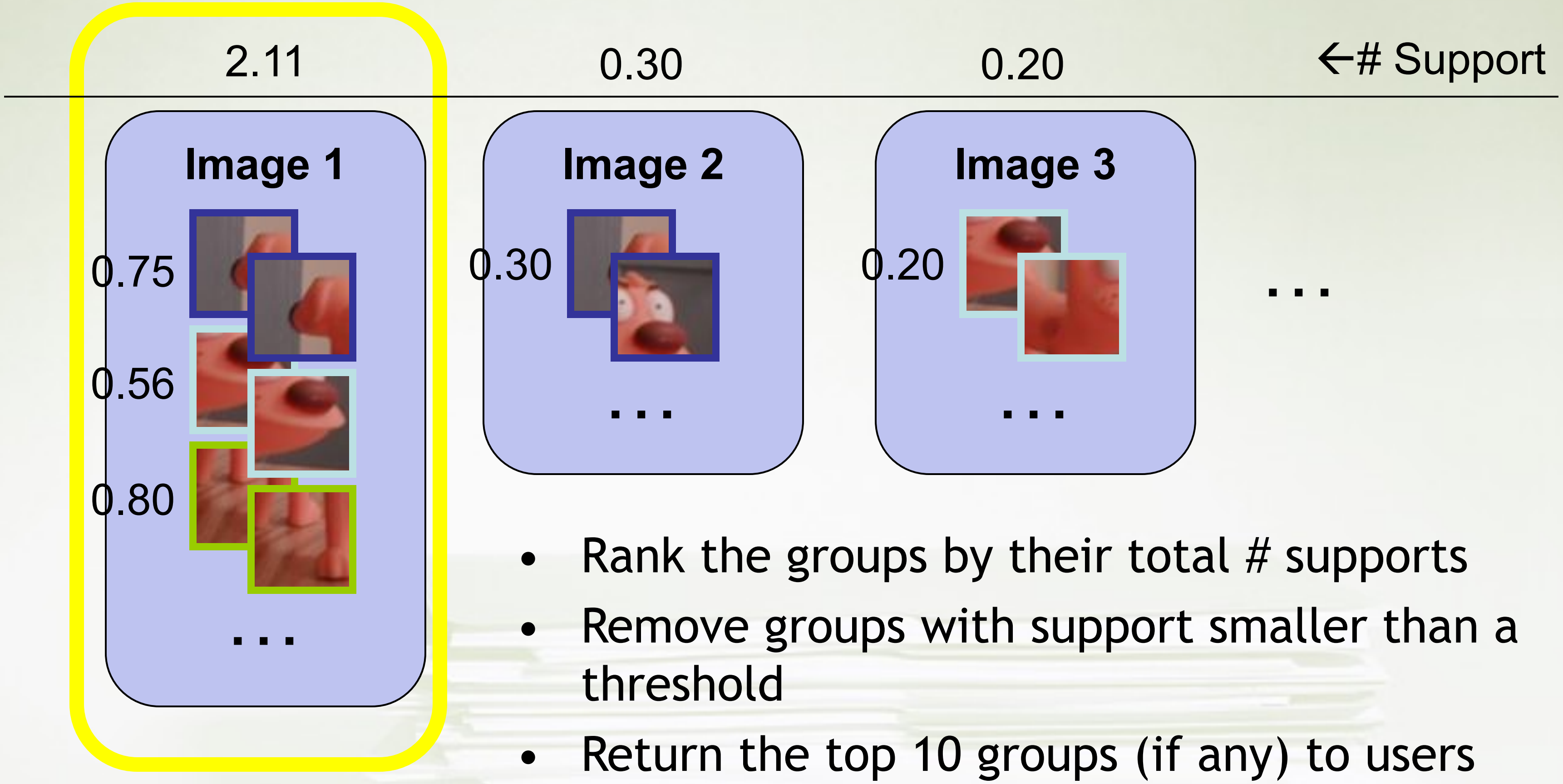
- We can compute **A** with 3 matching pairs (x_0,b_0), (x_1,b_1) & (x_2,b_2)

$$\begin{bmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{01} \\ a_{02} \\ a_{10} \\ a_{11} \\ a_{12} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ v_0 \\ v_1 \\ v_2 \end{bmatrix}$$

The RANSAC Step

1. Randomly pick 3 matching pairs
2. Compute **A**
3. Count the number of matches with L_2 distance between **Ax** and **b** smaller than a threshold (=> agree the transformation)
4. Loop to find the transformation with the greatest # support (= # matches)

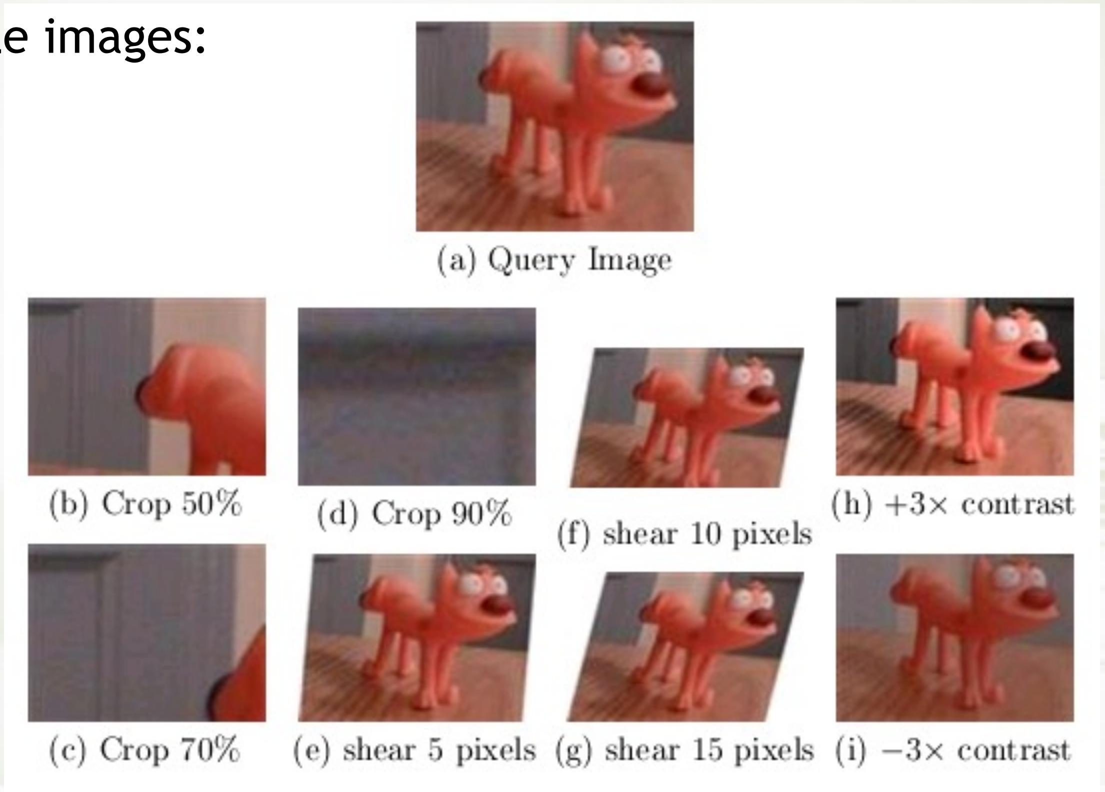
Database Query – Image Voting



- Rank the groups by their total # supports
- Remove groups with support smaller than a threshold
- Return the top 10 groups (if any) to users

Performance Evaluation

- We evaluate the performance of our system using a feature database containing 1 million of keypoints from 1200 images
- Images in the database are the transformed versions of the query images
- All of the transformations we chosen are **difficult** for IND detection
- Example images:



Performance Evaluation – Orientation Verification

- Evaluation Metrics
  - **Correct match** is a match between a query image and any one of its transformed versions in the database

$$recall = \frac{number\ of\ correct\ matches}{total\ number\ of\ correct\ matches}$$

$$precision = \frac{number\ of\ correct\ matches}{total\ number\ of\ matches}$$

Parameter Name	Value
Matching strategy	K-NN matching strategy
R	350
K	10

Verification Schemes

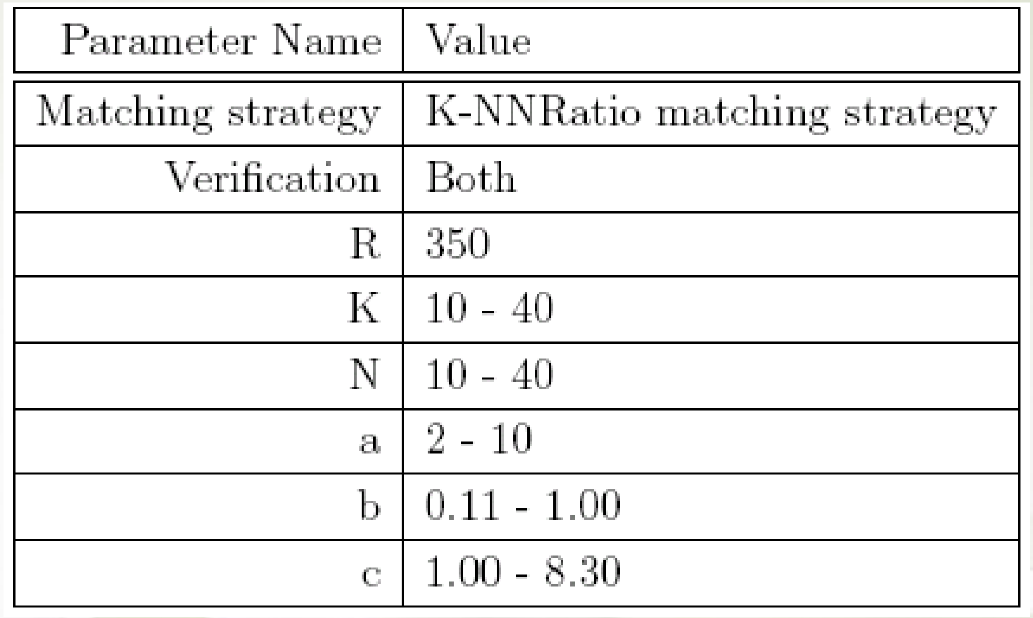
a) Orientation Verification
b) Affine Geometric Verification
c) Orientation Verification + Affine Geometric Verification

Verification	# correct matches	# false matches	recall	precision
(a)	975	471	81%	67%
(b)	1001	430	83%	70%
(c)	1011	301	84%	77%

+1% recall
+7% precision

Performance Evaluation – K-NNRatio

- Determine parameters a, b, and c by trying different values:



Parameter Name	Value
Matching strategy	K-NNRatio matching strategy
Verification	Both
R	350
K	10 - 40
N	10 - 40
a	2 - 10
b	0.11 - 1.00
c	1.00 - 8.30

- We find that setting (a = 4, b = 0.2, and c = 4) gives the best result
- Under this setting:

	# correct	# false	recall	Precision
K-NN	1011	301	84%	77%
K-NNRatio	1040	177	87%	85%

+3% recall
+8% precision

Summary

- We have introduced the followings:
  - The IND detection system
  - The orientation verification process
  - The *k*-NNRatio matching strategy

Conclusions

- We presented
  - A scalable content-based image retrieval scheme
    - We suggested a parallel and distributed solution to overcome the memory consumption problem of LSH
    - Extensive empirical evaluation is performed on large dataset
  - An IND detection scheme to remove the near-duplicate images during image crawling process
    - The scheme integrated powerful feature detector, descriptor, new matching strategy, and new verification process

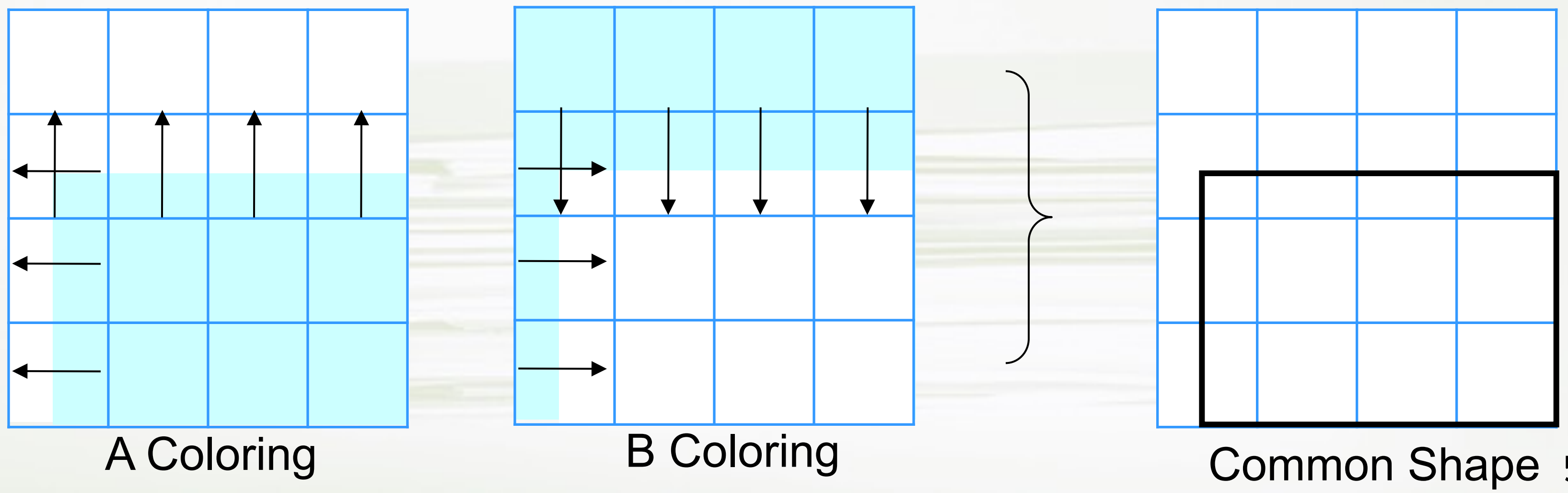
Q & A

***Appendix*:**

Shape-SIFT Feature Descriptor

Background and Object Color Invariance

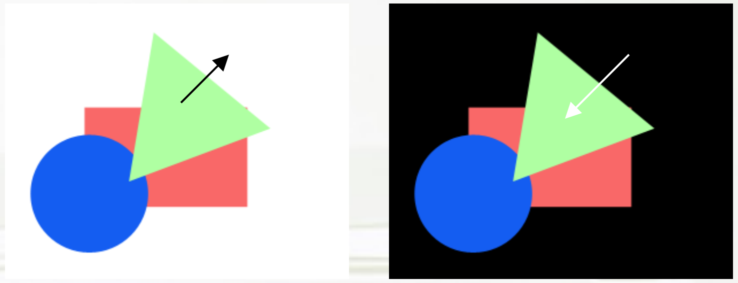
- We have tested that SIFT performs better than GLOH and PCA-SIFT in term of recall rate
- However, SIFT is not robust to changing background and object color
- SIFT is a histogram of gradient orientation of pixels within a 16 x 16 local window
- Gradient orientations may flip in direction when the colors of feature change
  - E.g. Entries in 0^o bin moved to 180^o bin (great diff. during matching)



Background and Object Color Invariance

- Orientation flipping happens when background and object change in color / luminance
- More precisely, orientation flipping occurs along the contour where color on either/both sides change
- Contour of object is an important clue for recognizing object. Thus, correctly describing features near contour is a key to success in object recognition

Background color change



Object color change

Our Feature Descriptor

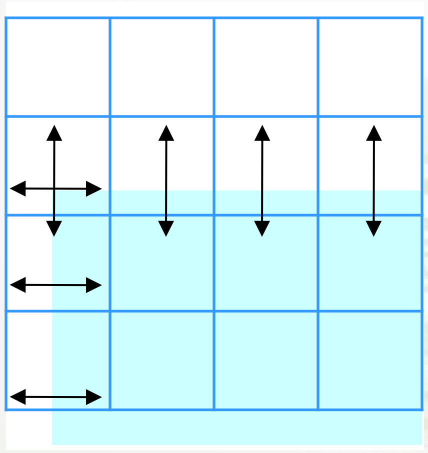
- Orientation Assignment
  - For each sample in the Gaussian smoothed image L, L(x,y), the orientation θ(x,y) is computed as follow:

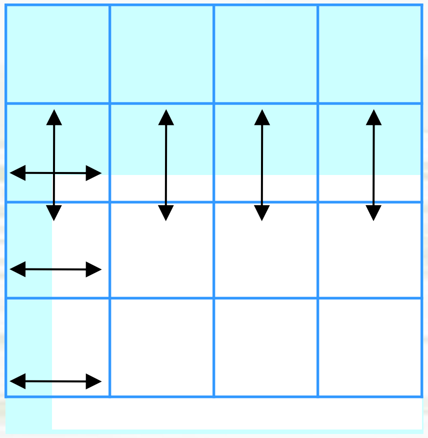
$$\theta(x,y) = \tan^{-1}\left|\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right|$$

Vertical pixel intensity difference

Horizontal pixel intensity difference

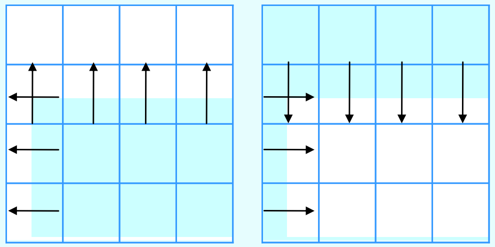
Don't care which side has higher intensity





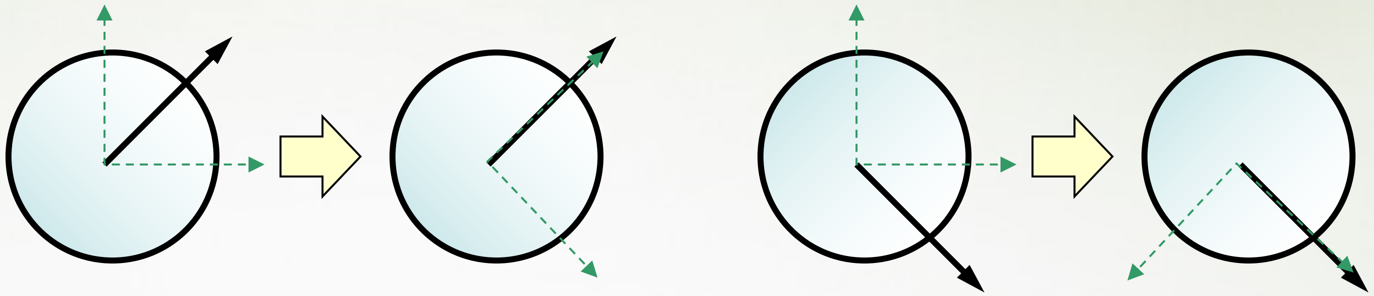
cf. SIFT Orientation Assignment

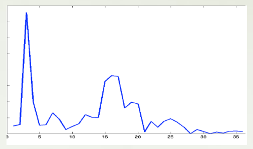
$$\theta(x,y) = \tan^{-1}\left[\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right]$$



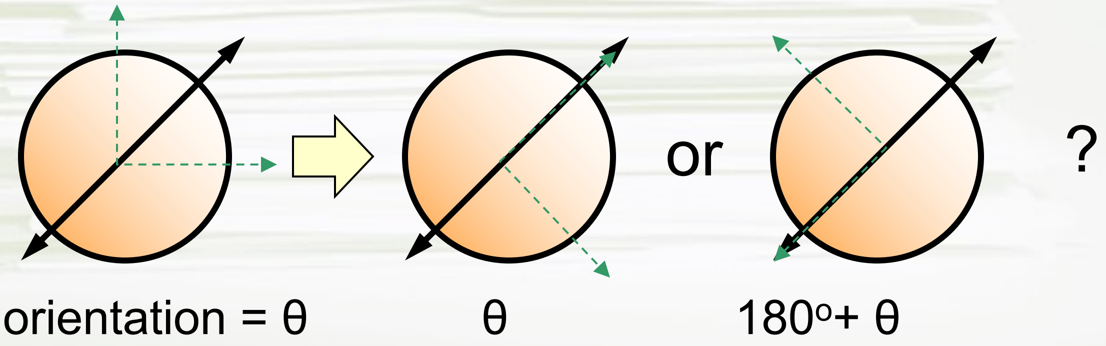
Canonical Orientation Determination

- Keypoint Descriptor
  - In order to achieve rotation invariance, the coordinates of the descriptor is rotated according to the **dominant orientation**





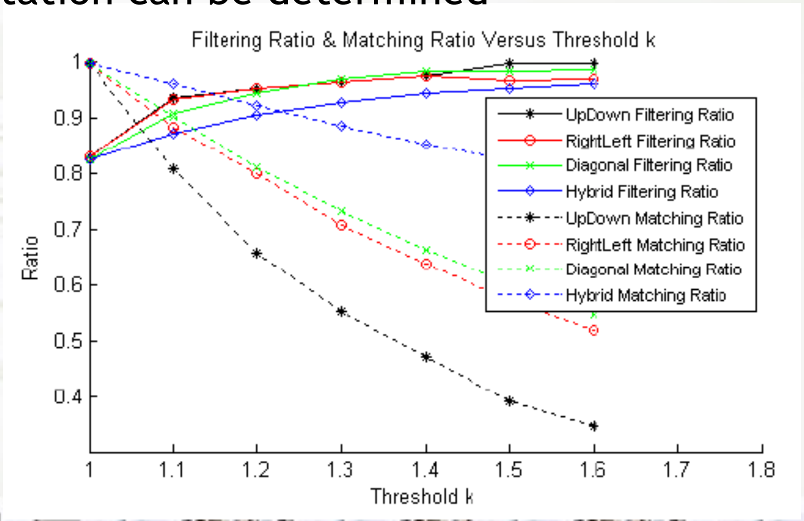
  - However, we have limited the orientation assignment to [0^o,180^o] and thus the dominant orientation also lies within [0^o,180^o]
  - This causes ambiguity in rotating the coordinates of the descriptor



Canonical Orientation Determination

- We proposed two approaches to tackle this problem:
  - Duplication
    - Generate two descriptors
    - One with canonical orientation θ and the other with 180º+ θ
  - *By the distribution of peak orientation*
    - Determine the dominant orientation using a feature's orientation dependent statistic value
    - Insignificant adverse effect to matching feature
    - Over 75% of feature's orientation can be determined

0	1	2	3
1	0	1	2
2	1	0	1
3	2	1	0



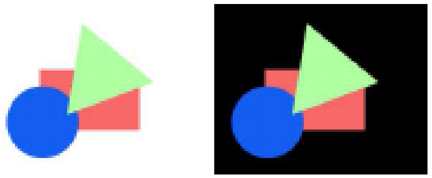
SSIFT-64 and SIFT-128

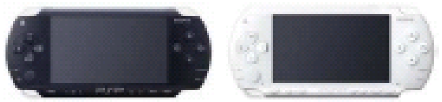
- SSIFT-64 descriptor is computed by **concatenating 8 4-bin orientation histograms** over the feature region together
- Problem
  - SSIFT-64 is more invariant to background and object color change
  - it is inevitably less distinctive
- Solution
  - Append 8 more 4-bin orientation histograms to SSIFT-64 to produce SSIFT-128
  - Each of these histograms contains north, east, south and west orientation bins, covering 360 degree range of orientation
  - We adopted a matching mechanism such that the performance of our descriptor on changing background and object color is retained while increasing the recall and precision on other cases

Matching Mechanism SSIFT-128

- Matching mechanism for SSIFT-128 descriptor
  - Find the best match *$match_{64}$* using the first 64 elements of SSIFT-128 (SSIFT-64) and calculate the distance ratio *dr_{64}*
  - Then refine the match using also last 64 elements and calculate the distance ratio dr_{128}
  - If $dr_{64} > dr_{128}$
    - Best match = $match_{64}$
  - Else
    - Best match = $match_{128}$
  - Experiments show that the overall performance of SSIFT-128 is better than SSIFT-64

Performance Evaluation

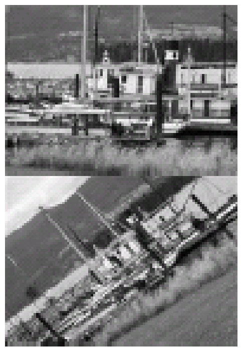

(a) Polygons


(b) PSP

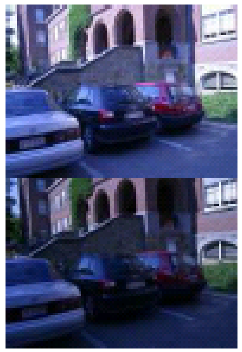
(c) dragon

(e) basmati

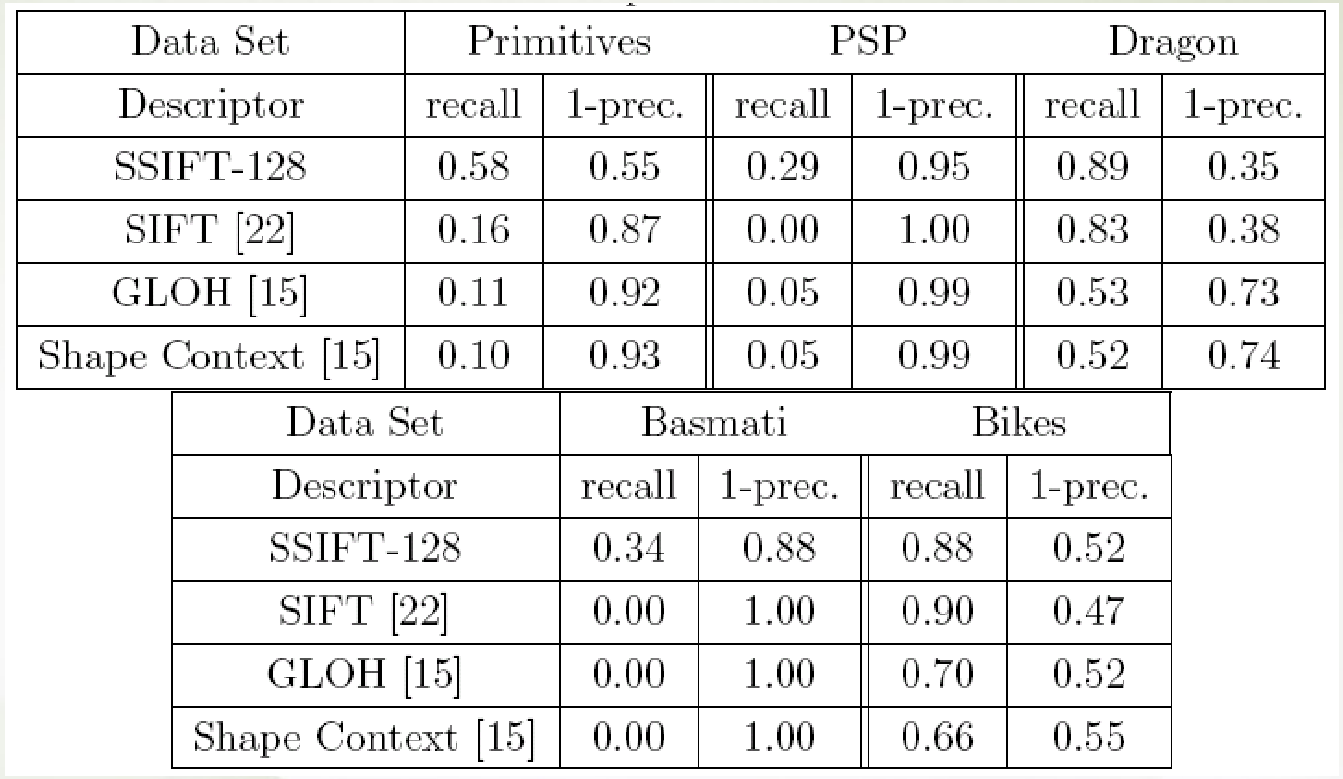
(f) bikes


(g) boat

(h) graf


(i) leuven

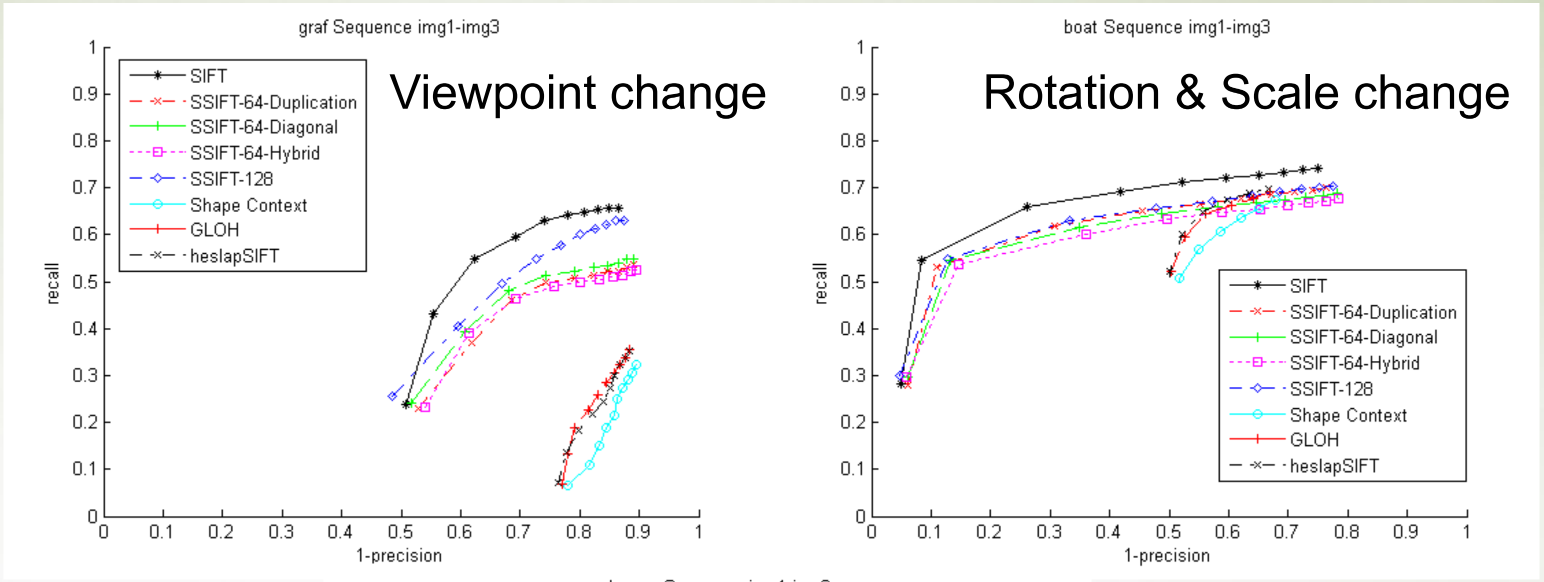
Performance Evaluation

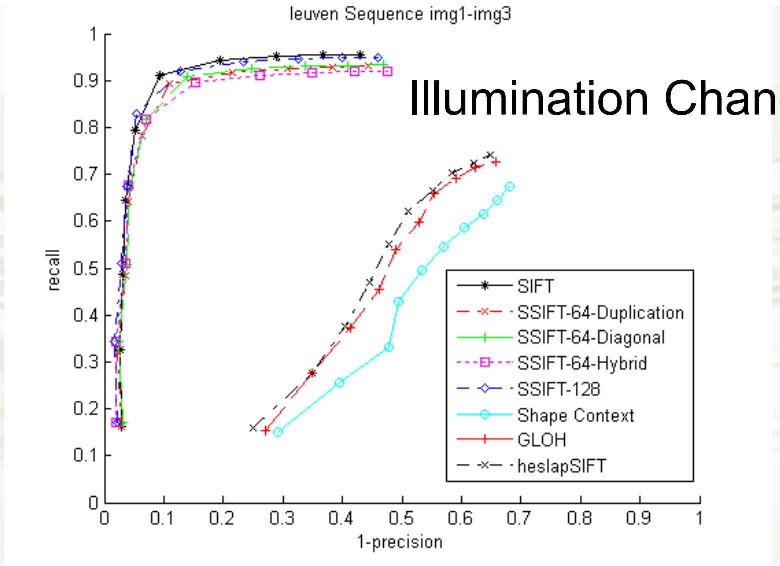


Data Set	Primitives		PSP		Dragon	
Descriptor	recall	1-prec.	recall	1-prec.	recall	1-prec.
SSIFT-128	0.58	0.55	0.29	0.95	0.89	0.35
SIFT [22]	0.16	0.87	0.00	1.00	0.83	0.38
GLOH [15]	0.11	0.92	0.05	0.99	0.53	0.73
Shape Context [15]	0.10	0.93	0.05	0.99	0.52	0.74

Data Set	Basmati		Bikes	
Descriptor	recall	1-prec.	recall	1-prec.
SSIFT-128	0.34	0.88	0.88	0.52
SIFT [22]	0.00	1.00	0.90	0.47
GLOH [15]	0.00	1.00	0.70	0.52
Shape Context [15]	0.00	1.00	0.66	0.55

Performance Evaluation


graf Sequence img1-img3
Viewpoint change
boat Sequence img1-img3
Rotation & Scale change
recall
1-precision
SIFT
SSIFT-64-Duplication
SSIFT-64-Diagonal
SSIFT-64-Hybrid
SSIFT-128
Shape Context
GLOH
heslapSIFT


leuven Sequence img1-img3
Illumination Change
recall
1-precision
SIFT
SSIFT-64-Duplication
SSIFT-64-Diagonal
SSIFT-64-Hybrid
SSIFT-128
Shape Context
GLOH
heslapSIFT