Storage for MMIR

Thomas Breuel

UniKL

covering

- image formats
- bandwidth / CPU tradeoffs
- file system
- relational databases
- scientific databases
- common schemas

Image Formats

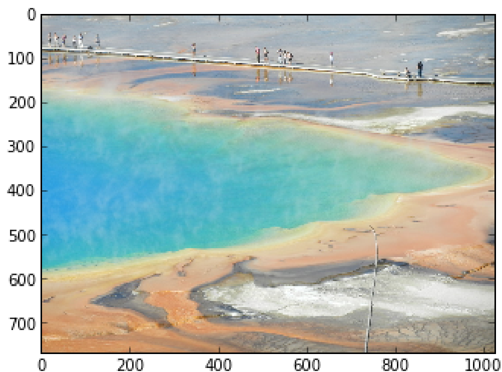
image formats

- raw (various): $w \times h$, 16 bpp, image as it comes out of the camera
- uncompressed (PPM): $w \times h \times 3$, usually 8 bpp (sometimes 16 bpp)
- lossless compression (PNG)
- lossy compression (JPEG, JPEG2000)

important tools

- ImageMagick, GNU Magic (convert, etc.)
- SciPy
- PIL (Python imaging library)
- Python exiv2 (EXIF metadata in Python)

```
iname = "raw_images/DSCN0243.JPG"
image = imread(iname)
imshow(image)
```



getting metadata

```
import pyexiv2
metadata = pyexiv2.ImageMetadata(iname)
metadata.read()
for key in metadata.exif_keys[::10]:
    print key,"\t",metadata[key]
```

```
Exif.Image.ImageDescription  <Exif.Image.ImageDescription [Ascii] =
Exif.Image.ExifTag  <Exif.Image.ExifTag [Long] = 252>
Exif.Photo.CompressedBitsPerPixel  <Exif.Photo.CompressedBitsPerPixel [Rational] = 
Exif.Nikon3.Version  <Exif.Nikon3.Version [Undefined] = 0 2 0 0>
Exif.Nikon3.ISOSelection  <Exif.Nikon3.ISOSelection [Ascii] = AUTO   >
...
Exif.Nikon3.ShotInfo  <Exif.Nikon3.ShotInfo [Undefined] = (Binary value suppressed)
Exif.Nikon3.0x00b5  <Exif.Nikon3.0x00b5 [Short] = 4098>
Exif.Photo.SceneType  <Exif.Photo.SceneType [Undefined] = 1>
Exif.Photo.Sharpness  <Exif.Photo.Sharpness [Short] = 0>
```

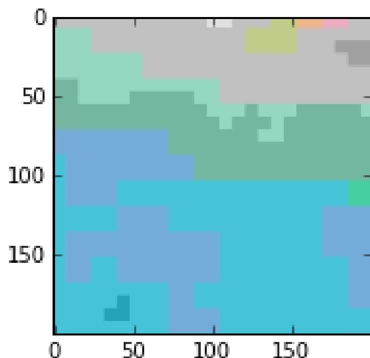
image conversion and size

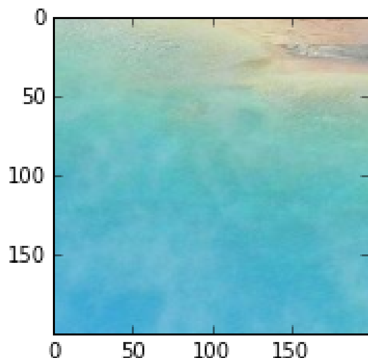
```
1
2 !convert raw_images/DSCN0243.JPG DSCN0243.ppm
3 !convert DSCN0243.ppm DSCN0243.png
4 !ls -lh raw_images/DSCN0243.JPG DSCN0243.*
5 !rm DSCN0243.ppm DSCN0243.png
```

```
-rw-r--r-- 1 tmb tmb 916K Oct 31 04:17 DSCN0243.png
-rw-r--r-- 1 tmb tmb 2.3M Oct 31 04:17 DSCN0243.ppm
-rw-r--r-- 1 tmb tmb 201K Oct 30 14:33 raw_images/DSCN0243.JPG
```

JPEG quality settings

```
from PIL import Image
im = Image.open(iname)
im.save("temp.jpg",quality=1)
subplot(121); imshow(imread("temp.jpg")[200:400,200:400])
im.save("temp.jpg",quality=80)
subplot(122); imshow(imread("temp.jpg")[200:400,200:400])
```





Remember JPEG Compression

- divide image into 8×8 squares
- perform DCT for each square
- quantize and select coefficients
- linearize through scanning
- compress

in-memory compression requires streams

```python

import StringIO
stream = StringIO.StringIO()
im.save(stream,'jpeg')
compressed = stream.getvalue()
print len(compressed)
```

84770

useful in-memory compression functions

```
import PIL,StringIO

def cjpeg(image,quality=98):
    im = PIL.Image.fromarray(image)
    stream = StringIO.StringIO()
    im.save(stream,'jpeg',quality=quality)
    compressed = stream.getvalue()
    return compressed

def djpeg(compressed):
    stream = StringIO.StringIO(compressed)
    im = PIL.Image.open(stream)
    return numpy.array(im)
```

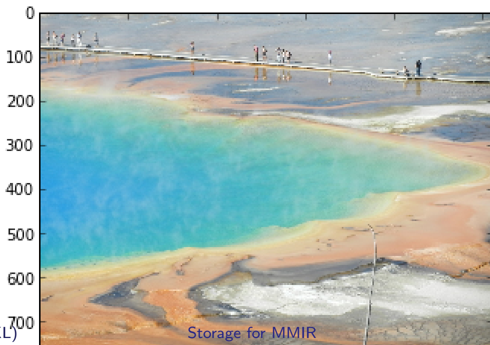
increasing performance

- compression reduces I/O (less data to move)
- compression has some overhead
- overall, storing compressed images is faster
- devices already use compression for other data types
- special image compression algorithms are more effective

testing in-memory compression

```
1
2 compressed = cjpeg(imread(iname))
3 decompressed = djpeg(compressed)
4 imshow(decompressed)
5 print prod(decompressed.shape),len(compressed)
```

```
2359296 275947
```



Icons

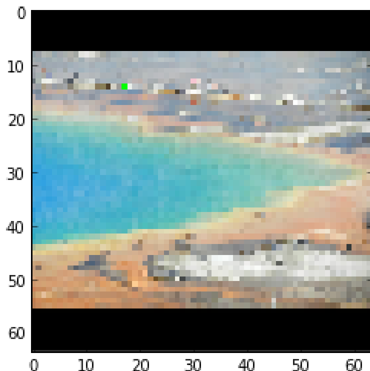
Both for display and for analysis, it's useful to store *icons* of the images being indexed.

- fixed size and centered (square)
- may be stored uncompressed (if smaller than the size where compression makes sense)
- 8 bpp

```python
from scipy.ndimage import interpolation

def make_icon(image,size=(256,256)):
    if amax(image)>1.0: image = image/255.0
    assert amin(image)>=0.0 and amax(image)<=1.0
    h,w = size
    ih,iw,d = image.shape
    scale = min(h*1.0/ih,w*1.0/iw)
    scaled = interpolation.zoom(image,(scale,scale,1))
    sh,sw,_ = scaled.shape
    result = zeros((h,w,3))
    dh,dw = h//2-sh//2,w//2-sw//2
    result[dh:dh+sh,dw:dw+sw,:] = scaled
    return array(result*255,'B')
```

```
image = imread(iname)
icon = make_icon(image,(64,64))
print icon.shape,icon.dtype
imshow(icon,interpolation='none')
```

```
(64, 64, 3) uint8
```



Checksums and Duplicate Detection

```python
import md5

def md5file(fname):
    m = md5.new()
    with open(fname,"rb") as stream:
        while 1:
            s = stream.read(4096)
            if s=="": break
            m.update(s)
    return m.hexdigest()
```

```
md5file(iname)
```

```
'5640ebaac01fc22241874189d85dff9a'
```

Relational Databases

opening the database, creating the table

```
import sqlite3
!rm -f sample.db
dbname = "sample.db"
db = sqlite3.connect(dbname)
c = db.cursor()
c.execute("create table images (chk text primary key,path text,w integer,h integer,jpeg blob,icon blob,time date)")
```

```
<sqlite3.Cursor at 0x5aecf10>
```

what we are storing

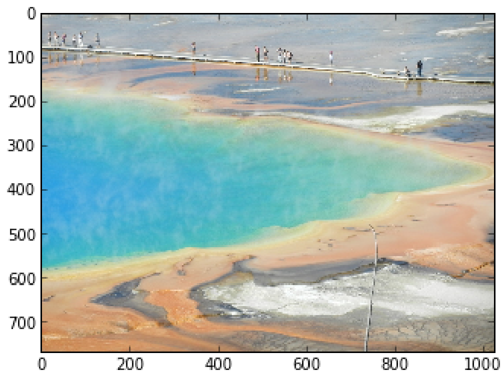
```
1
2 chk = md5file(iname)     # MD5 checksum of the file
3 image = imread(iname)    # JPEG-compressed image
4 jpeg = cjpeg(image)
5 h,w,d = image.shape      # dimensions of the image
6 icon = make_icon(image) # an icon (uncompressed)
7 when = datetime.datetime.now()
```

inserting the data into the database

```
# note the use of ‘buffer‘ for blobs
values = (chk,iname,w,h,buffer(jpeg),buffer(icon),when)
c.execute("insert into images values (?,?,?,?,?,?,?)",values)
db.commit()
```

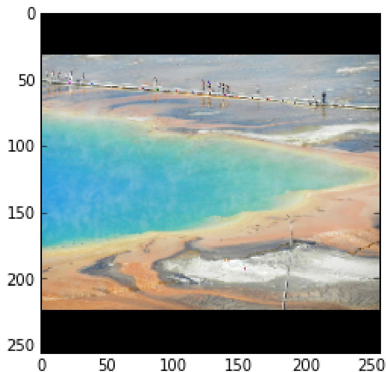
retrieving a jpeg image

```
1
2 result = list(c.execute("select jpeg from images limit 1"))
3 imshow(djpeg(result[0][0]))
```



retrieving an icon

```
1
2 result = list(c.execute("select icon from images limit 1"))
3 imshow(array(result[0][0],'B').reshape(256,256,3))
```



getting metadata

```
import pyexiv2
from datetime import datetime

def image2db(iname,db,c):
    chk = md5file(iname)     # MD5 checksum of the file
    if list(c.execute("select chk from images where chk=?",(chk,)))
        !=[]:
        c.execute("update images set path=? where chk=?",(iname,chk
            ))
        db.commit()
        return False

    # get metadata
    metadata = pyexiv2.ImageMetadata(iname)
    metadata.read()
    if "Exif.Photo.DateTimeOriginal" in metadata:
        when = metadata["Exif.Photo.DateTimeOriginal"].value
    else:
        when = datetime.now()

    # get content data
    image = imread(iname)    # JPEG-compressed image
    jpeg = cjpeg(image)
    h,w,d = image.shape      # dimensions of the image
```

clear out the database

```
1
2 c.execute("delete␣from␣images")
3 db.commit()
```

```python
import glob

for i,iname in enumerate(sorted(glob.glob("raw_images/*.JPG"))):
    if i%10==0: print i,iname
    if not image2db(iname,db,c):
        print iname,"already in the database"
```

```
0 raw_images/DSCN0149.JPG
10 raw_images/DSCN0204.JPG
20 raw_images/DSCN0254.JPG
30 raw_images/DSCN0286.JPG
40 raw_images/DSCN0320.JPG
...
170 raw_images/P1010409.JPG
180 raw_images/P1010458.JPG
190 raw_images/P1010496.JPG
200 raw_images/P1010536.JPG
```

make sure the checksum-based duplicate detection is actually working

```
1
2 for i,iname in enumerate(sorted(glob.glob("raw_images/*.JPG")[:5]))
      :
3     if i%10==0: print i,iname
4     if not image2db(iname,db,c):
5         print iname,"already␣in␣the␣database"
```

```
0 raw_images/DSCN0225.JPG
raw_images/DSCN0225.JPG already in the database
raw_images/DSCN0557.JPG already in the database
raw_images/DSCN0642.JPG already in the database
raw_images/DSCN0649.JPG already in the database
raw_images/P1010456.JPG already in the database
```

an SQL query to determine all the different size images

```
1
2 print list(c.execute("select count(*) from images"))
3 print list(c.execute("select distinct w,h from images"))
```

```
[(207,)]
[(1024, 179), (1024, 768), (768, 1024), (1024, 328), (1024, 421), (1024, 164), (102
```

cleaning up and size comparisons

```
1
2 db.close()
3 !du -h raw_images
4 !ls -lh sample.db
```

```
49M raw_images
-rw-r--r-- 1 tmb tmb 106M Oct 31 04:19 sample.db
```

Scientific Databases

HDF5

HDF5 is a database for large scientific datasets in array format.

- no transactions
- limited concurrent access
- arrays as primitives
- very fast random access
- special compression routines for numerical data, floating point
- similar to "column databases" and NoSQL databases
- integrates directly with NumPy/SciPy (Python)
- supported in a wide range of languages, including C++ and Fortran

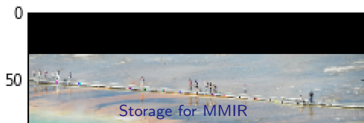
```
1 import tables
2 from tables import openFile,Filters,Int32Atom,Float32Atom,Int64Atom
      ,UInt8Atom,VLStringAtom
```

Persistent Arrays

HDF5 effectively implements *persistent arrays* or *memory mapped arrays* (the actual implementation may or may not use memory mapping, but it looks that way from Python):

```
hdf = tables.openFile("temp.h5","w")
hdf.createEArray("/","icons",UInt8Atom(),shape=[0,256,256,3])
hdf.root.icons.append([icon])
hdf.close()

hdf = tables.openFile("temp.h5","r+")
print len(hdf.root.icons)
imshow(hdf.root.icons[0])
hdf.close()
```

1



VLArrays, Strings

You can have variable length arrays of variable length strings. Strings are special and a bit different from numerical arrays. You can also use VLArrays with 1D numerical arrays, giving a ragged array. If you want to store variable sized 2D images, you need to store the dimensions in a separate column and reshape on access.

```
hdf = tables.openFile("sample6.h5","w")
hdf.createVLArray("/","chk",VLStringAtom())
hdf.root.chk.append(u"hello")
hdf.root.chk.append("world")
hdf.root.chk[0:2]
hdf.close()
```

store all the icons from the image database in HDF5

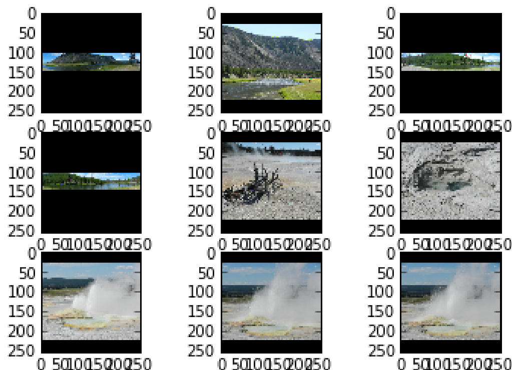
```
hdf = tables.openFile("sample.h5","w")
hdf.createVLArray("/","chk",VLStringAtom())
hdf.createEArray("/","icons",UInt8Atom(),shape=[0,256,256,3],
    filters=tables.Filters(9))

db = sqlite3.connect(dbname)
c = db.cursor()
rows = c.execute("select chk,icon from images")
for row in rows:
    chk = str(row[0])
    icon = array(row[1],'B').reshape(256,256,3)
    hdf.root.chk.append(chk)
    hdf.root.icons.append([icon])

db.close()
hdf.close()
```

```
!ls -lh sample.h5
!h5ls sample.h5
```

```
-rw-r--r-- 1 tmb tmb 20M Oct 31 04:20 sample.h5
chk                      Dataset {207/Inf}
icons                    Dataset {207/Inf, 256, 256, 3}
```

quickly accessing HDF5 content

```
with tables.openFile("sample.h5","r") as hdf:
    for i in range(9):
        subplot(3,3,i+1)
        imshow(hdf.root.icons[i])
```



Summary - Storage

Storage

- images can be stored compressed or raw
- compressed storage increases speed because CPU is cheaper than I/O
- metadata in EXIF format for images
- storage on file system, in relational databases, in scientific databases
- relational databases offer transactions
- scientific database are optimized for large array storage
- in-memory compression needed and usually available via “string streams”