Memory

Main memory – large storage media, CPU can access directly.

Secondary storage – extension of main memory, provides large nonvolatile storage capacity.

Magnetic disks – rigid metal or glass platters covered with magnetic recording material

Disk surface is logically divided into tracks, which are subdivided into sectors.

The disk controller determines the logical interaction between the device and the computer.
Memory Hierarchy

- registers
- cache
- main memory
- electronic disk
- magnetic disk
- optical disk
- magnetic tapes

faster, more volatility, cost more
SATA 2 / UDMA

A sector is a sub-division of a track of a magnetic hard disk or optical disc.

A sector stores a fixed amount of data.

Modern discs have fixed capacity, variable size sectors.

DMA direct memory access  
block memory transfer

SATA 2 ("representative" values)  
2.4 Gbit/s transfer rates  
8 Mb buffer Size
example hard disk

Drive specification ST9500420AS
Formatted Gbytes (512 bytes/sector) 500
Guaranteed sectors 976,773,168
Bytes per sector 512
Physical read/write heads 4
Discs 2
Cache (Mbytes) 16
Recording density in BPI (bits/inch max) 1,490k
Track density TPI (tracks/inch max) 265k
Areal density (Gbits/inch2 max) 394
Spindle speed (RPM) 7,200
Average latency (msec) 4.17
Internal transfer rate (Gbits/sec max) 1.23
I/O data transfer rate (Gbits/sec max) 3.0
ATA data-transfer modes supported
   SATA 1.0,
   Serial ATA Revision 2.6
   PIO modes 0–4
   Multiword DMA modes 0–2
   Ultra DMA modes 0–6
Serialization

OOP APIs provide for object persistence – the serialization of objects or collections to secondary storage.

```java
anObjectStream.writeObject(anObject);
...
anObject = (Object cast) anObjectStream.readObject();
```

The object to be persisted must implement the Serializable interface

The object to be persisted must mark all nonserializable fields `transient`

JFC collections can be serialized to persist between application sessions.

see posted example (and “record” file)

SerializedArrayListDemo.java
SerializedDataObject.java
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class ASerializableClass {
    ...
    ASerializableClass asObj; // assume exists
    ObjectOutputStream fOut = null;
    ...
    try {
        fOut = new ObjectOutputStream(
            new FileOutputStream(filename));
        fOut.writeObject(asObj);
        // asObj = (ASerializableObject)
        inFile.readObject();
        fOut.close(); } 
    catch (IOException ex) { 
        ex.printStackTrace(); } 
    ...
}
Random access file

Conceptual random access file (RAF) = collection of “records” stored in data blocks.

Block access – indexed (seek), read, and write blocks

\[
\text{buffer.readBlock(dataFile, i)}
\]
\[
\text{buffer.writeBlock(dataFile,i)}
\]

Blocking factor (BF) = data records / block

record \( i \) is in block \( i / BF \)

RAF is “an array” of blocks where each block has a “bucket capacity” of BF

\[
B_0 \quad B_1 \quad B_2 \quad B_i \quad B_{n-1}
\]
\[
R_0 \quad R_1 \quad R_2 \quad \quad \quad \quad R_{n-2} \quad R_{n-1}
\]
Java Random Access File

Java “records” are objects
Java RandomAccessFile methods

RandomAccessFile (String fileName, String mode)
RandomAccessFile (File file, String mode)
   mode { “r”, “rw”, “rws”, “rwd” }

void write(int b)
void write(byte [] b)
void writeFloat(float f)
void writeBytes(String s)

int read()    // reads a byte
int read(byte [])    // 0..255
int read(byte [] dst, int offset, int length)
int readInt()  // reads 4 bytes 32 bit int

int position()    // returns byte offset in file
void seek(long pos)    // makes offset pos in file
class RAFDataObject implements Serializable {
    // see Java's Serializable docs
    static final long serialVersionUID = 4242L;
    int id, age;
    String name, gender; // stored in Byte[]
    // larger than needed.
    public static final int byteBufferSize = 40;
    ...
}

RAFDataObject size in bytes
4 id
4 age
4 name length
8 name's largest value "sunshine"
4 gender length
7 gender's largest value "unknown"
----
31 bytes
name's byte size = 4 + 8 bytes

To write strings to a “fixed length” data block
allocate space for the length of the string and
then convert the string to ByteBuffer.
see examples:
StringBuilderSize.java
RAFdemo.java
RAFDataObject.java
public class RAFdemo {
    RAFDataObject RAFdObj;
    int byteBufferSize;
    RandomAccessFile inRAF;
    RandomAccessFile outRAF;

    try {
        outRAF = new RandomAccessFile(fileName, "rw");
        for (int i = 0; i < n/2; i++) {
            outRAF.seek(index[i] * byteBufferSize);
            RAFdObj = new RAFDataObject(aRandom.nextInt(1000));
            System.out.println(index[i] + " " + RAFdObj);
            outRAF.write(RAFdObj.getStore().array());
        }
        System.out.println(fileName + " size = " + outRAF.length());
        outRAF.close();
    }
    catch(IOException ex) { ex.printStackTrace(); }
    ...
}
External Hashing

If implementing hashing for a file just like internal hashing with open addressing and linear probe.

File store capacity = (numberOfRecords / loadFactor) * recordSize

\[ fsc = \left( \frac{nor}{lf} \right) \times rs \]

if \( nor = 10,000 \) \( lf = 2/3 \) \( rs = 250 \) bytes

\[ fsc = 15,000 \times 250 = 3,750,000 \text{ bytes} \]

If the file block on our system is 512 bytes

blocking factor, \( bf = \left[ \frac{sectorSize}{rs} \right] = 2 \)

\[ \left[ \frac{fsc}{sectorSize \div bf} \right] = 3663 \text{ sectors} \]

units of transfer for data file

When “record” size is multiple of sector (file block) then seeks become block I/O requests.
hashed access

\[
DataObject \ aDataObject \\
int rs = DataObject.byteBufferSize \\
byte [] byteArray = new byte[rs]
\]

// hash with open addressing, linear probe
public DataObject hashGet(key) {
    boolean stopProbe = false
    hashedIndex = hashFn(key) // ≈ key % FileStoreCapacity
    do {
        RAFinFile.seek(hashedIndex * rs) // byte offset
        RAFinFile.read(byteArray)
        aDataObject = new DataObject(byteArray)
        if (aDataObject.getKey() == key)
            stopProbe = true // found key, stop probe
        else if (aDataObject.getKey() == -1)
            aDataObject = null // key does not exist
            stopProbe = true // stop probe
        else // continue probe in cluster
            hashIndex = (hashIndex + 1) % FileStoreCapacity
    } while (! stopProbe)
    return aDataObject
}