**UNIT-IV**

1. **Why raid? Define and discuss the concept in detail.**

**Or**

**Explain the raid levels in detail** (NOV 2016)

**Raid: redundant Arrays of Independent Disks**

* disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
* high capacity and high speed by using multiple disks in parallel, and
* high reliability by storing data redundantly, so that data can be recovered even if a disk fails
* The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail.
* E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
* Techniques for using redundancy to avoid data loss are critical with large numbers of disks
* Originally a cost-effective alternative to large, expensive disks
* I in RAID originally stood for ``inexpensive’’
* Today RAIDs are used for their higher reliability and bandwidth.
* The “I” is interpreted as independent

**Improvement of Reliability via Redundancy**

* **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
* E.g., **Mirroring** (or **shadowing**)
* Duplicate every disk. Logical disk consists of two physical disks.
* Every write is carried out on both disks
* Reads can take place from either disk
* If one disk in a pair fails, data still available in the other
* Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
* Probability of combined event is very small
* Except for dependent failure modes such as fire or building collapse or electrical power surges
* Mean time to data loss depends on mean time to failure, and mean time to repair
* E.g. MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500\*106 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

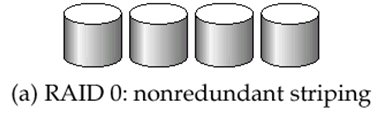
**Improvement in Performance via Parallelism**

* Two main goals of parallelism in a disk system:
* Load balance multiple small accesses to increase throughput
* Parallelize large accesses to reduce response time.
* Improve transfer rate by striping data across multiple disks.
* **Bit-level striping** – split the bits of each byte across multiple disks
* In an array of eight disks, write bit *i* of each byte to disk *i.*
* Each access can read data at eight times the rate of a single disk.
* But seek/access time worse than for a single disk
* Bit level striping is not used much any more
* **Block-level striping** – with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1
* Requests for different blocks can run in parallel if the blocks reside on different disks
* A request for a long sequence of blocks can utilize all disks in parallel

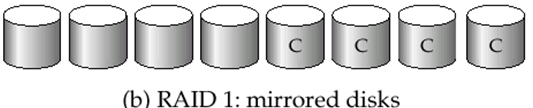
**RAID Levels**

Schemes to provide redundancy at lower cost by using disk striping combined with parity bits. Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics

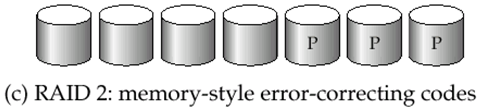
* **RAID Level 0**: Block striping; non-redundant.
* Used in high-performance applications where data lost is not critical



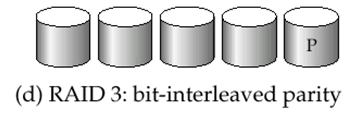
* **RAID Level 1**: Mirrored disks with block striping
* Offers best write performance.
* Popular for applications such as storing log files in a database system.



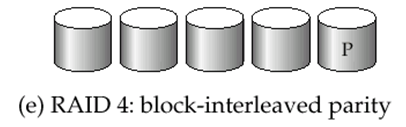
* **RAID Level 1**: Mirrored disks with block striping
* Offers best write performance.
* Popular for applications such as storing log files in a database system.



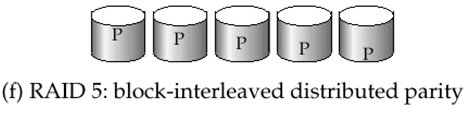
* **RAID Level 3**: Bit Interleaved Parity
* a single parity bit is enough for error correction, not just detection, since we know which disk has failed
* When writing data, corresponding parity bits must also be computed and written to a parity bit disk
* To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk).
* RAID level 3 is as good as level 2, but is less expensive in the number of extra disks, so level 2 is not used in practice.
* Level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every disk, and thus reduces the storage overhead.
* Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.

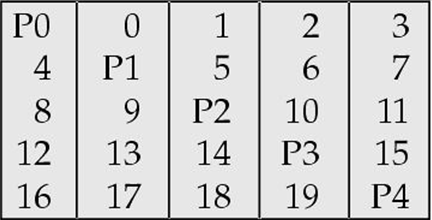


* **RAID Level 4:** Block-Interleaved Parity;
* uses block level striping, and keeps a parity block on a separate disk for corresponding blocks from *N* other disks.
* When writing data block, corresponding block of parity bits must also be computed and written to parity disk
* To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.
* Provides higher I/O rates for independent block reads than Level 3
* block read goes to a single disk, so blocks stored on different disks can be read in parallel
* Provides high transfer rates for reads of multiple blocks than no striping
* Before writing a block, parity data must be computed Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
* Or by re-computing the parity value using the new values of blocks corresponding to the parity block
* More efficient for writing large amounts of data sequentially
* Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

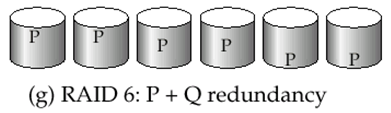


* **RAID Level 5:** Block-Interleaved Distributed Parity;
* Partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk.
* E.g., with 5 disks, parity block for *n*th set of blocks is stored on disk (*n mod* 5) + 1, with the data blocks stored on the other 4disks.
* Higher I/O rates than Level 4.
* Block writes occur in parallel if the blocks and their parity blocks are on different disks.
* Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.





* **RAID Level 6**: P+Q Redundancy
* P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
* Better reliability than Level 5 at a higher cost; not used as widely.



**Choice of RAID Level:**

* Factors in choosing RAID level
* Monetary cost
* Performance: Number of I/O operations per second, and bandwidth during normal operation
* Performance during failure
* Performance during rebuild of failed disk
* Including time taken to rebuild failed disk
* RAID 0 is used only when data safety is not important
* E.g. data can be recovered quickly from other sources
* Level 2 and 4 never used since they are subsumed by 3 and 5
* Level 3 is not used since bit-striping
* forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
* Level 6 is rarely used since levels 1 and 5 offer adequate safety for almost all applications. So competition is between 1 and 5 only
* Level 1 provides much better write performance than level 5
* Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
* Level 1 preferred for high update environments such as log disks
* Level 1 had higher storage cost than level 5
* disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
* I/O requirements have increased greatly, e.g. for Web servers
* When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity. So there is often no extra monetary cost for Level 1.
* Level 5 is preferred for applications with low update rate, and large amounts of data
* Level 1 is preferred for all other applications

**Hardware Issues:**

* **Software RAID**: RAID implementations done entirely in software, with no special hardware support
* **Hardware RAID**: RAID implementations with special hardware
* Use nonvolatile RAM to record writes that are being executed
* Beware: power failure during write can result in corrupted disk
* E.g. failure after writing one block but before writing the second in a mirrored system
* Such corrupted data must be detected when power is restored
* Recovery from corruption is similar to recovery from failed disk
* NVRAM helps to efficiently detected potentially corrupted blocks Otherwise all blocks of disk must be read and compared with mirror/parity block
* **Hot swapping**: replacement of disk while system is running, without power down Supported by some hardware RAID systems,
* reduces time to recovery, and improves availability greatly
* Many systems maintain spare disks which are kept online, and used as replacements for failed disks immediately on detection of failure
* Reduces time to recovery greatly
* Many hardware RAID systems ensure that a single point of failure will not stop the functioning of the system by using Redundant power supplies with battery backup
* Multiple controllers and multiple interconnections to guard against controller/interconnection failures
* **Log disk** – a disk devoted to writing a sequential log of block updates
* Used exactly like nonvolatile RAM
* Write to log disk is very fast since no seeks are required
* No need for special hardware (NV-RAM)
* File systems typically reorder writes to disk to improve performance
* **Journaling file systems** write data in safe order to NV-RAM or log disk
* Reordering without journaling: risk of corruption of file system data

1. **DISCUSS FILE ORGANIZATION IN DETAIL ( MAY 2014) (dec 2016)**

A file is organized logically as a sequence of records.

* These records are mapped on to disk blocks.
* The blocks are of a fixed size are determined by the physical properties of the disk and by the operating system, the record size may vary.
* We have to consider ways os representing logical data models in terms of files.
* The following is the example of a file containing the account records.

|  |  |  |  |
| --- | --- | --- | --- |
| Record 0 | A-102 | Perryridge | 400 |
| Record 1 | A-305 | Round Hill | 350 |
| Record 2 | A-215 | Mianus | 700 |
| Record 3 | A-101 | Downtown | 500 |
| Record 4 | A-222 | Redwood | 700 |
| Record 5 | A-201 | Perryridge | 900 |
| Record 6 | A-217 | Brighton | 750 |
| Record 7 | A-110 | Downtown | 600 |
| Record 8 | A-218 | Perryridge | 700 |

* In a relational database, tuples of distinct relations are generally of different sizes.
* One approach to mapping the database to files is to use several files and to store records of only one fixed length in any given files. It is called as**fixed length records.**
* An alternative is to structure our files is accommodating multiple lengths for records. It is called as **variable-length records***.*

**FIXED-LENGTH RECORDS**

* Store record *i* starting from byte *n \* (i –* 1), where *n* is the size of each record.
* Record access is simple but records may cross blocks
* **Modification**: do not allow records to cross block boundaries
* Though, we have two problems with this simple approach:
* It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of this file, or we must have way of marking deleted records so that they can be ignored.
* Unless the block happens to be a multiple of 40(which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accessses to read or write such a record.
* When a record is deleted, we could move the record that came after it into the space formally occupied by the deleted record and so on, until every record following the deleted record has been moved ahead.(fig-1)

******

**Fig.1** ***With record 2 deleted and all records moved.***

* It might be easier simply to move the final record of the file into the space occupied by the deleted record.(fig-2)

****

**Fig.2 *with record 2 deleted and final record moved***

* A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.
* At the beginning of the file, we allocate a certain number of bytes as a **file header.**
* The header will contain a variety of information about the file.
* For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record and so on.
* These deleted records form a linked list, which is often referred to as a **free list.** *(*shown in the fig 3).

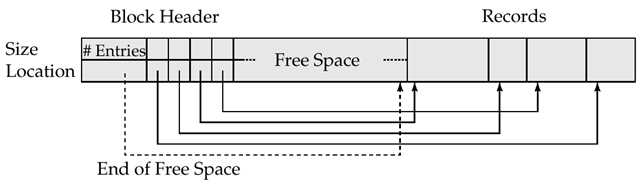
****

**Fig.3 *With free list after deletion of records 1, 4, and 6.***

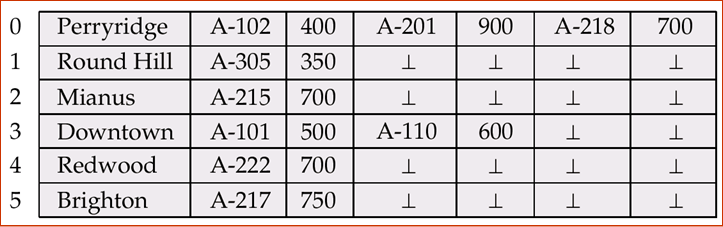
* On insertion of a new record, we use the record pointed to by the header.
* We change the header pointer to point to the next available record. If no space is available, we add the new file to the end of the record.

**Variable-Length Records**

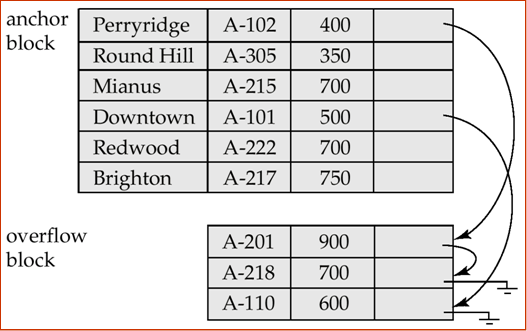
* variable-length records arise in database systems in several ways:
* Storage of multiple record types in a file.
* Record types that allow variable lengths for one or more fields.
* Record types that allow repeating fields.
* Different techniques for implementing variable-length records exist. They are:
* byte-string representation.
* fixed-length representation.
* **Byte string representation**
* Attach an *end-of-record* (⊥) control character to the end of each record
* Difficulty with deletion
* Difficulty with growth



* Slotted page header contains:
* number of record entries
* end of free space in the block
* location and size of each record
* Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
* Pointers should not point directly to record — instead they should point to the entry for the record in header
* **Fixed-length representation:**
* reserved space
* pointers
* **Reserved space** – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol



* **Pointer method**
* A variable-length record is represented by a list of fixed-length records, chained together via pointers.
* Can be used even if the maximum record length is not known
* Disadvantage to pointer structure; space is wasted in all records except the first in a chain.
* Solution is to allow two kinds of block in file:
* Anchor block – contains the first records of chain
* Overflow block – contains records other than those that are the first records of chairs.



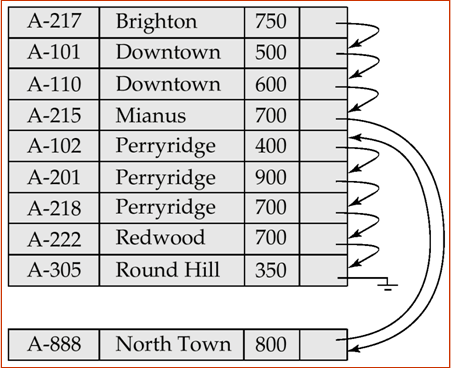


**Organization of Records in Files**

* **Heap** – a record can be placed anywhere in the file where there is space
* **Sequential** – store records in sequential order, based on the value of the search key of each record
* **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
* Records of each relation may be stored in a separate file. In a **clustering file organization**  records of several different relations can be stored in the same file

**Sequential File Organization**

* Suitable for applications that require sequential processing of the entire file
* The records in the file are ordered by a search-key

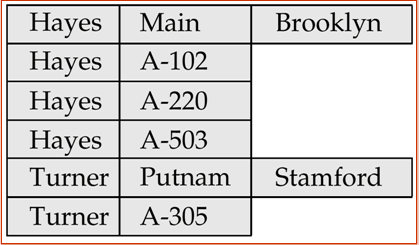
 

1. Deletion – use pointer chains
2. Insertion –locate the position where the record is to be inserted
3. if there is free space insert there
4. if no free space, insert the record in an overflow block
5. In either case, pointer chain must be updated

* Need to reorganize the file from time to time to restore sequential order

**Clustering File Organization**

* Simple file structure stores each relation in a separate file
* Can instead store several relations in one file using a **clustering** file organization
* E.g., clustering organization of *customer* and *depositor:*



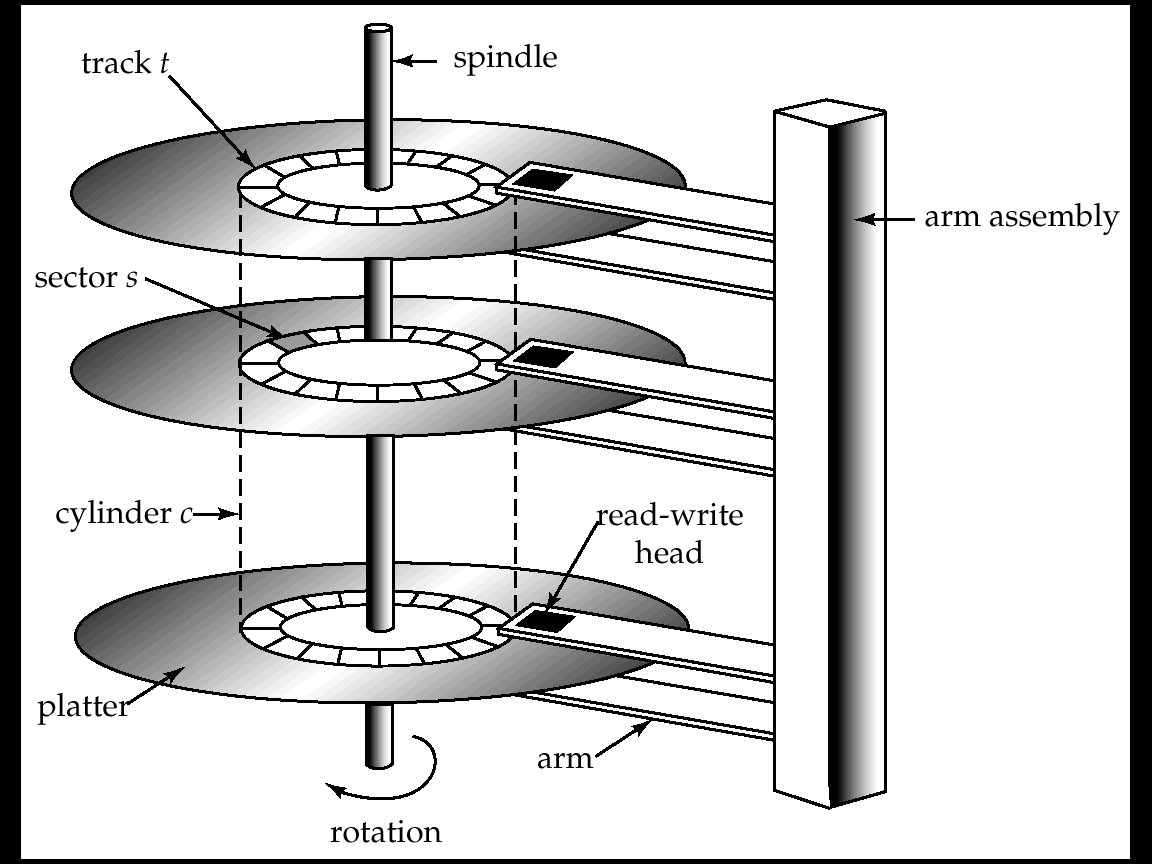
* good for queries involving depositor customer, and for queries involving one single customer and his accounts
* bad for queries involving only customer
* results in variable size records

**3.DISCUSS IN DETAIL THE MAGNETIC DISKS. (MAY 2009)**

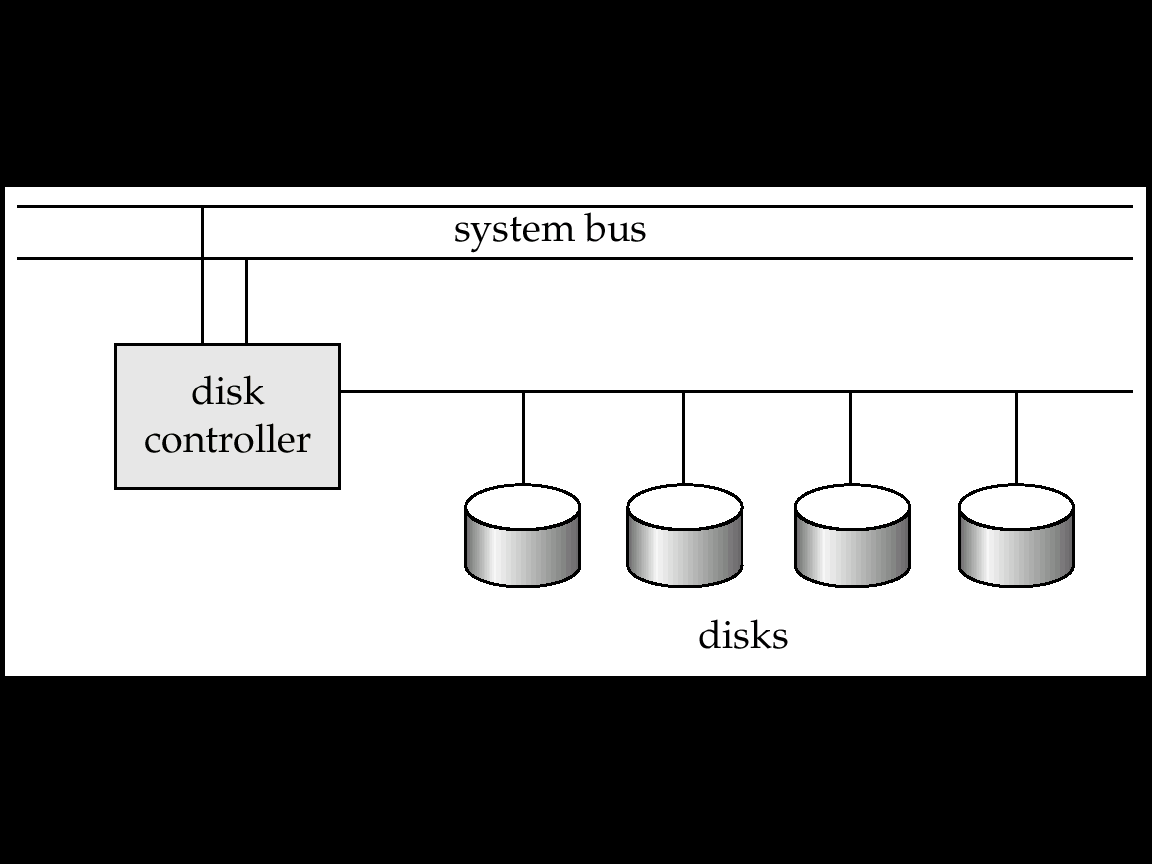
Magnetic disks provide the bulk of secondary storage for modern computer systems. Disk capacities have been growing at over 50 percent per year, but the storage requirements of large applications have also been growing very fast, in some cases even faster than the growth rate of disk capacities. A large database may require hundreds of disks.

**Physical Characteristics of Disks**

Physically, disks are relatively simple. Each disk **platter** has a flat circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass and are covered (usually on both sides) with magnetic recording material. We call such magnetic disks **hard disks**, to distinguish them from **floppy disks**, which are made from flexible material.



* **Read-write head**
* Positioned very close to the platter surface (almost touching it)
* Reads or writes magnetically encoded information.
* Surface of platter divided into circular **tracks**
* Over 16,000 tracks per platter on typical hard disks
* Each track is divided into **sectors.**
* A sector is the smallest unit of data that can be read or written.
* Sector size typically 512 bytes
* Typical sectors per track: 200 (on inner tracks) to 400 (on outer tracks)
* To read/write a sector
* disk arm swings to position head on right track
* platter spins continually; data is read/written as sector passes under head
* Head-disk assemblies
* multiple disk platters on a single spindle (typically 2 to 4)
* one head per platter, mounted on a common arm.
* **Cylinder** *i*consists of *i*th track of all the platters
* Earlier generation disks were susceptible to head-crashes
* Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
* Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
* **Disk controller** – interfaces between the computer system and the disk drive hardware.
* accepts high-level commands to read or write a sector
* initiates actions such as moving the disk arm to the right track and actually reading or writing the data
* Computes and attaches **checksums** to each sector to verify that data is read back correctly
* If data is corrupted, with very high probability stored checksum won’t match recomputed checksum
* Ensures successful writing by reading back sector after writing it
* Performs remapping of bad sectors



* Multiple disks connected to a computer system through a controller
* Controllers functionality (checksum, bad sector remapping) often carried out by individual disks; reduces load on controller
* Disk interface standards families
* ATA (AT adaptor) range of standards
* SCSI (Small Computer System Interconnect) range of standards
* Several variants of each standard (different speeds and capabilities)

**Performance Measures of Disks**

* **Access time** – the time it takes from when a read or write request is issued to when data transfer begins.
* **Seek time** – time it takes to reposition the arm over the correct track.
* **Average seek time** is 1/2 the worst case seek time.
* Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
* 4 to 10 milliseconds on typical disks
* **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
* Average latency is 1/2 of the worst case latency.
* 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
* **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
* 4 to 8 MB per second is typical
* Multiple disks may share a controller, so rate that controller can handle is also important
* E.g. ATA-5: 66 MB/second, SCSI-3: 40 MB/s
* Fiber Channel: 256 MB/s
* **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
* Typically 3 to 5 years
* Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 30,000 to 1,200,000 hours for a new disk
* E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
* MTTF decreases as disk ages

**Optimization of Disk-Block Access**

* **Block** – a contiguous sequence of sectors from a single track
* data is transferred between disk and main memory in blocks
* sizes range from 512 bytes to several kilobytes
* Smaller blocks: more transfers from disk
* Larger blocks: more space wasted due to partially filled blocks
* Typical block sizes today range from 4 to 16 kilobytes
* **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
* **elevator algorithm** : move disk arm in one direction (from outer to inner tracks or vice versa), processing next request in that direction, till no more requests in that direction, then reverse direction and repeat
* **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
* E.g. Store related information on the same or nearby cylinders.
* Files may get **fragmented** over time
* E.g. if data is inserted to/deleted from the file
* Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
* Sequential access to a fragmented file results in increased disk arm movement
* Some systems have utilities to defragment the file system, in order to speed up file access
* **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
* Non-volatile RAM: battery backed up RAM or flash memory
* Even if power fails, the data is safe and will be written to disk when power returns
* Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
* Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
* Writes can be reordered to minimize disk arm movement

1. **EXPLAIN THE HIERARCHY OF VARIOUS STORAGE MEDIA (NOV 2009)** **(APR 2011)**

**Classification of Physical Storage Media**

* Speed with which data can be accessed
* Cost per unit of data
* Reliability
* data loss on power failure or system crash
* physical failure of the storage device
* Can differentiate storage into:
* **volatile storage:** loses contents when power is switched off
* **Nonvolatile storage**: Contents persist even when power is switched off. Includes secondary and tertiary storage, as well as Battery backed up main memory

**Physical Storage Media**

**Cache**

Fastest and most costly form of storage; volatile; managed by the computer system hardware (Note: “Cache” is pronounced as “cash”)

**Main memory**

* Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10–9 seconds)
* Generally too small (or too expensive) to store the entire database capacities of up to a few Gigabytes widely used currently. Capacities have gone up and per byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
* **Volatile:** contents of main memory are usually lost if a power failure or system crash occurs.

**Flash memory**

* Data survives power failure
* Data can be written at a location only once, but location can be erased and written to again
* Can support only a limited number (10K – 1M) of write/erase cycles.
* Erasing of memory has to be done to an entire bank of memory
* Reads are roughly as fast as main memory
* But writes are slow (few microseconds), erase is slower
* NOR Flash
* Fast reads, very slow erase, lower capacity
* Used to store program code in many embedded devices
* NAND Flash
* Page-at-a-time read/write, multi-page erase
* High capacity (several GB)
* Widely used as data storage mechanism in portable devices

**Magnetic-disk**

* Data is stored on spinning disk, and read/written magnetically
* Primary medium for the long-term storage of data; typically stores entire database.
* Data must be moved from disk to main memory for access, and written back for storage
* **direct-access** – possible to read data on disk in any order, unlike magnetic tape
* Survives power failures and system crashes
  + - * disk failure can destroy data: is rare but does happen

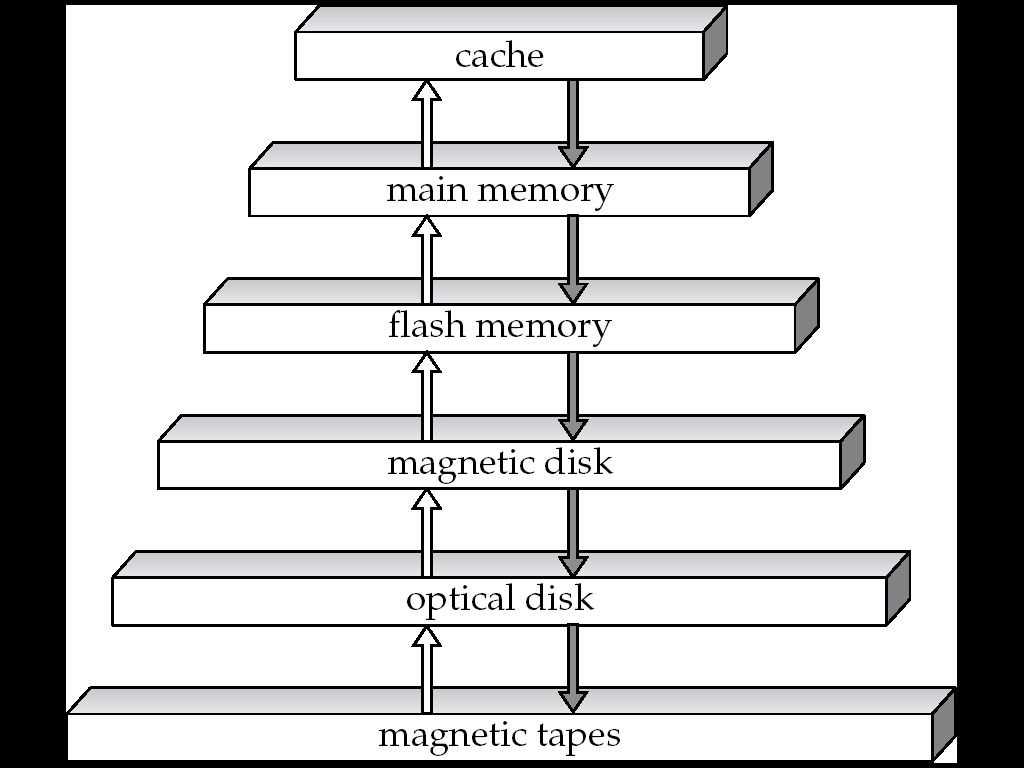
**Optical storage**

* non-volatile, data is read optically from a spinning disk using a laser
* CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
* Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
* Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
* Reads and writes are slower than with magnetic disk
* **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

**Tape storage**

* non-volatile, used primarily for backup (to recover from disk failure), and for archival data
* **sequential-access** – much slower than disk
* very high capacity (40 to 300 GB tapes available)
* tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
* Tape jukeboxes available for storing massive amounts of data hundreds of terabytes (1 terabyte = 109 bytes) to even a petabyte (1 petabyte = 1012 bytes)

**Storage Hierarchy**



**Primary storage**: Fastest media but volatile (cache, main memory).

**Secondary storage**: next level in hierarchy, non-volatile, moderately fast access time also called on-line storage

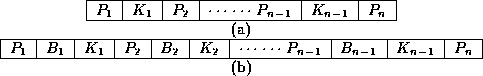
E.g. flash memory, magnetic disks

**Tertiary storage**: lowest level in hierarchy, non-volatile, slow access time also called off-line storage

E.g. magnetic tape, optical storage

# 5.Explain about B-TREE and Dynamic Hashing( MAY 2012) ( APR 2015)

1. B-tree indices are similar to B tex2html_wrap_inline829-tree indices.
   * Difference is that B-tree eliminates the redundant storage of search key values.
   * In B tex2html_wrap_inline829-tree of Figure 11.11, some search key values appear twice.
   * A corresponding B-tree of Figure 11.18 allows search key values to appear only once.
   * Thus we can store the index in less space.



**Figure 11.8:**   Leaf and nonleaf node of a B-tree.

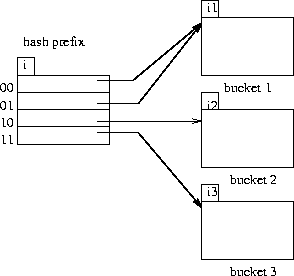
1. **Advantages:**

* Generally, the structural simplicity of B tex2html_wrap_inline829-tree is preferred.
* Lack of redundant storage (but only marginally different).
* Some searches are faster (key may be in non-leaf node).

1. **Disadvantages:**
   * Leaf and non-leaf nodes are of different size (complicates storage)
   * Deletion may occur in a non-leaf node (more complicated)

# DYNAMIC HASHING

1. As the database grows over time, we have three options:
   * Choose hash function based on current file size. Get performance degradation as file grows.
   * Choose hash function based on anticipated file size. Space is wasted initially.
   * Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.
2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.
   * **Extendable hashing** is one form of dynamic hashing.
   * Extendable hashing splits and coalesces buckets as database size changes.
   * This imposes some performance overhead, but space efficiency is maintained.
   * As reorganization is on one bucket at a time, overhead is acceptably low.
3. How does it work?

  
**Figure 12.4**   General extendable hash structure.

* + We choose a hash function that is uniform and random that generates values over a relatively large range.
  + Range is *b*-bit binary integers (typically b=32).
  + tex2html_wrap_inline1075 is over 4 billion, so we don't generate that many buckets!
  + Instead we create buckets on demand, and do not use all *b* bits of the hash initially.

tex2html_wrap_inline1081.

* + The *i* bits are used as an offset into a table of bucket addresses.
  + Value of *i* grows and shrinks with the database.
  + Figure 12.4 shows an extendable hash structure.
  + Note that the *i* appearing over the bucket address table tells how many bits are required to determine the correct bucket.
  + It may be the case that several entries point to the same bucket.
  + All such entries will have a common hash prefix, but the length of this prefix may be less than *i*.
  + So we give each bucket an integer giving the length of the common hash prefix.
  + The integer associated with bucket j is shown as 
  + Number of bucket entries pointing to bucket *j* is then tex2html_wrap_inline1095.

1. To find the bucket containing search key value tex2html_wrap_inline1097:
   * Compute tex2html_wrap_inline1099.
   * Take the first *i* high order bits of tex2html_wrap_inline1099.
   * Look at the corresponding table entry for this *i*-bit string.
   * Follow the bucket pointer in the table entry.
2. We now look at insertions in an extendable hashing scheme.
   * Follow the same procedure for lookup, ending up in some bucket *j*.
   * If there is room in the bucket, insert information and insert record in the file.
   * If the bucket is full, we must split the bucket, and redistribute the records.
   * If bucket is split we may need to increase the number of bits we use in the hash.
3. Two cases exist:

1. If tex2html_wrap_inline1109, then only one entry in the bucket address table points to bucket *j*.

* + Then we need to increase the size of the bucket address table so that we can include pointers to the two buckets that result from splitting bucket *j*.
  + We increment *i* by one, thus considering more of the hash, and doubling the size of the bucket address table.
  + Each entry is replaced by two entries, each containing original value.
  + Now two entries in bucket address table point to bucket j.
  + We allocate a new bucket *z*, and set the second pointer to point to *z*.
  + Set tex2html_wrap_inline1091and tex2html_wrap_inline1123to *i*.
  + Rehash all records in bucket *j* which are put in either *j* or *z*.
  + Now insert new record.
  + It is remotely possible, but unlikely, that the new hash will still put all of the records in one bucket.
  + If so, split again and increment *i* again.

2. If tex2html_wrap_inline1135, then more than one entry in the bucket address table points to bucket *j*.

* + Then we can split bucket *j* without increasing the size of the bucket address table
  + Note that all entries that point to bucket *j* correspond to hash prefixes that have the same value on the leftmost tex2html_wrap_inline1091bits.
  + We allocate a new bucket *z*, and set tex2html_wrap_inline1091and tex2html_wrap_inline1123to the original tex2html_wrap_inline1091value plus 1.
  + Now adjust entries in the bucket address table that previously pointed to bucket *j*.
  + Leave the first half pointing to bucket *j*, and make the rest point to bucket *z*.
  + Rehash each record in bucket *j* as before.
  + Reattempt new insert.

1. Note that in both cases we only need to rehash records in bucket *j*.
2. Deletion of records is similar. Buckets may have to be coalesced, and bucket address table may have to be halved.
3. Insertion is illustrated for the example deposit file of Figure 12.25



* + 32-bit hash values on bname are shown in Figure 12.26



. An initial empty hash structure is shown in Figure 12.27.



* + We insert records one by one.
  + We (unrealistically) assume that a bucket can only hold 2 records, in order to illustrate both situations described.
  + As we insert the Perryridge and Round Hill records, this first bucket becomes full.
  + When we insert the next record (Downtown), we must split the bucket.
  + Since tex2html_wrap_inline1163, we need to increase the number of bits we use from the hash.
  + We now use 1 bit, allowing us tex2html_wrap_inline1165buckets.
  + This makes us double the size of the bucket address table to two entries.
  + We split the bucket, placing the records whose search key hash begins with 1 in the new bucket, and those with a 0 in the old bucket (Figure 11.23).
  + Next we attempt to insert the Redwood record, and find it hashes to 1.
  + That bucket is full, and tex2html_wrap_inline1167.
  + So we must split that bucket, increasing the number of bits we must use to 2.
  + This necessitates doubling the bucket address table again to four entries (Figure 12.28).



* + We rehash the entries in the old bucket.
  + We continue on for the deposit records of Figure 12.25, obtaining the extendable hash structure of Figure 12.31

.

1. **Advantages:**
   * Extendable hashing provides performance that does not degrade as the file grows.
   * Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.
2. **Disadvantages:**
   * Extra level of indirection in the bucket address table
   * Added complexity
3. How to manage multiple file access

So far we have seen sequential file organization and fetching of records based on the primary key or the column which is frequently used in the query. But in real life the query is not restricted to primary key or single column. It will have various needs to see the records in different combinations of columns. In such case indexes on primary/single column will degrade the performance of the query. Because the table will have huge number of records, and index on any one column will efficiently fetch all the records which satisfies the search criteria for indexed column. But the query is still not complete.

**SELECT \* FROM EMPLOYEE WHERE DEPT\_ID = 20 and SALARY = 5000;**

Since there is index on DEPT\_ID, fetching all the employees who are working for department 20 is quick. But it will return large number of employees. Out of those employees, it has to filter the employees with salary = 5000. But this is same as querying a very big table with one search criteria. Hence this will be time consuming part of the query.

In real life situations, no one likes waiting and so is the case with query. We need to think of some better options than having index on single column/ primary keys.

It still has to filter the data with other search criteria on which we do not have index. What would be the result? This fetch becomes slow, in total reducing the cost of the query.

For example consider employee table with huge number of employees in different department. Imagine we have to select all the employees whose salary is 5000 and works for department ID 20. Also, we have index created only on DEPT\_ID. The query to select the records is as follows:

**Multiple Key Accesses**

To handle above situation, multiple key accesses are introduced. Here, we use combination of two or more columns which are frequently queried to get index. In the above example, both DEPT\_ID and SALARY are clubbed into one index and are stored in the files. Now what happens when we fire above query? It filters both DEPT\_ID = 20 and SALARY = 5000 at a shot and returns the result.

This type of indexing works well when all the columns used in the index are involved in the query. In above example we have index on (DEPT\_ID, SALARY) and both the columns are used in the query. Hence it resulted quickly. But what will happen when any one of the column is used in the query? This index will not be used to fetch the record. Hence query becomes slower.

Imagine another case where we have queried like below. Here both the columns are used differently- we have queried to find the salary less than 5000, not equal to 5000. What will happen in this case?

**SELECT \* FROM EMPLOYEE WHERE DEPT\_ID = 20 and SALARY < 5000;**

This query will use the index and will fetch the result quickly. Reason is it has to find the address of the record in the file where DEPT\_ID = 20 and SALARY = 5000 and then it has to return all the records which are stored previous to this address. Hence the query becomes simple and faster.

Imagine another case for the same example, where DEPT\_ID <20 but SALARY = 5000 is used to fetch the record. What will happen in this case?

**SELECT \* FROM EMPLOYEE WHERE DEPT\_ID < 40 and SALARY = 5000;**

This query will not use the index. Why? The order of the columns in the index is (DEPT\_ID, SALARY).  The multiple key indexes work as below.

The condition (DEPT\_ID1, SALARY1) < (DEPT\_ID2, SALARY2) is used only if

**DEPT\_ID1 < DEPT\_ID2**

**OR**

**DEPT\_ID1 = DEPT\_ID2 AND SALARY1 <SALARY2**

In this example, we are checking for second condition, where we have DEPT\_ID <40. But SALARY < 5000 should have been used for the index to be used. But it is not the case in the query. That means, it should have only one condition DEPT\_ID <40 or both conditions like DEPT\_ID = 20 AND SALARY<5000 to use the index. This method of ordering in index is called **lexicographic ordering**.

Have a look at below index file to understand these examples. We can see that, records are first grouped based on DEPT\_ID and then on SALARY. If we have to select DEPT\_ID <40 alone, then all the records less than DEPT\_ID =40 address have to selected which easy and quick. We get all records in a sequence here. DEPT\_ID1 <DEPT\_ID2 condition holds well here.

If DEPT\_ID =20 but SALARY < 5000 condition is used, pointer will go to DEPT\_ID = 20 Section in the file and then fetch all the records less than 5000. The index is used in the query making it faster. Here, second conditions for lexicographic works.

What if we have to search DEPT\_ID <40 and SALARY = 5000, it is same as traversing the whole table. We will not be able to find all requested records at one place.

Similarly, if only one column in the index is used, say SALARY = 5000, then also searching is a full table scan. Below diagram clearly shows SALARY = 5000 is scattered in the memory file and have to traverse whole file to get the result. Hence it is not efficient.

1. **Explain about Physical Storage Media**

**Cache**

fastest and most costly form of storage; volatile;managed by the computer system hardware

(Note: “Cache” is pronounced as “cash”)

**Main memory**

* Fast access (10s to 100s of nanoseconds; 1 nanosecond =

10–9 seconds)

* Generally too small (or too expensive) to store the entire database

capacities of up to a few Gigabytes widely used currently

Capacities have gone up and per byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)

* **Volatile** — contents of main memory are usually lost if a power

failure or system crash occurs.

**Flash memory**

* Data survives power failure
* Data can be written at a location only once, but location can be erased and written to again
* Can support only a limited number (10K – 1M) of write/erase cycles.
* Erasing of memory has to be done to an entire bank of memory
  + - Reads are roughly as fast as main memory
    - But writes are slow (few microseconds), erase is slower
    - NOR Flash
* Fast reads, very slow erase, lower capacity
* Used to store program code in many embedded devices
  + - NAND Flash
* Page-at-a-time read/write, multi-page erase
* High capacity (several GB)
* Widely used as data storage mechanism in portable devices

**Magnetic-disk**

* Data is stored on spinning disk, and read/written magnetically
* Primary medium for the long-term storage of data; typically stores entire database.
* Data must be moved from disk to main memory for access, and written back for storage
* **direct-access** – possible to read data on disk in any order, unlike magnetic tape
* Survives power failures and system crashes
  + - * disk failure can destroy data: is rare but does happen

**Optical storage**

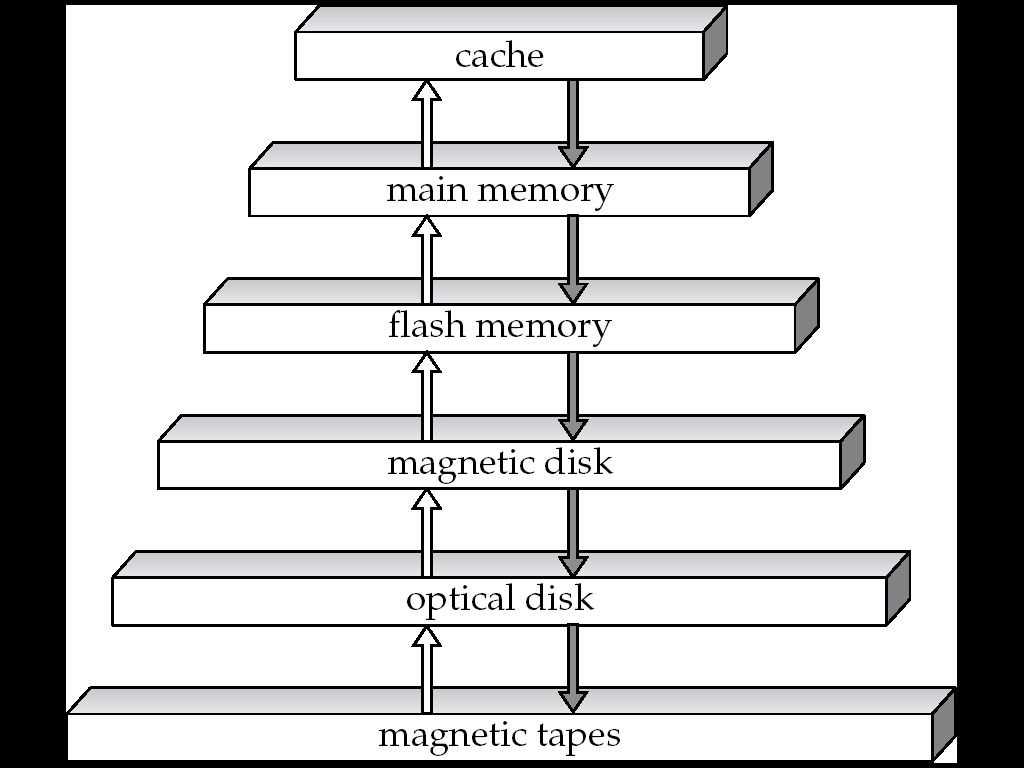
* non-volatile, data is read optically from a spinning disk using a laser
* CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
* Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
* Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
* Reads and writes are slower than with magnetic disk
* **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

**Tape storage**

* non-volatile, used primarily for backup (to recover from disk failure), and for archival data
* **sequential-access** – much slower than disk
* very high capacity (40 to 300 GB tapes available)
* tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
* Tape jukeboxes available for storing massive amounts of data

hundreds of terabytes (1 terabyte = 109 bytes) to even a petabyte (1 petabyte = 1012 bytes)

**Storage Hierarchy**



**primary storage**: Fastest media but volatile (cache, main memory).

**secondary storage**: next level in hierarchy, non-volatile, moderately fast access time also called on-line storage

E.g. flash memory, magnetic disks

**tertiary storage**: lowest level in hierarchy, non-volatile, slow access time also called off-line storage

E.g. magnetic tape, optical storage