UNIT IV

TRENDS IN DATABASE TECHNOLOGY

Classification o f Physical Storage Media

Based on Speed with which data can be accessed Based on Cost per unit of data Based on eliability Data loss on power failure or system crash Physical failure of the storage device H Based on life of storage Volatile storage: loses contents when power is switched off 4 Non-volatile storage:
Contents persist even when power is switched off. Includes secondary and tertiary storage, as well as batterbacked up main-memory.

Physical Storage Media

Cache

The fastest and most costly form of storage Volatile Managed by the computer system hardware.

Main memory
4 Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10–9 seconds) 4 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

Volatile — contents of main memory are usually lost if a power failure or system crash occurs.

Flash memory
Data survives power failureData can be written at a location only once, but location can be erased and written to again **Can support only a limited number 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 Cost per unit of storage roughly similar to main memory Widely used in embedded devices such as digital cameras Also known as EEPROM

Magnetic-disk
Data is stored on spinning disk,and read/written magneticallyPrimary medium for the long-term storage of data;typically stores entire database.Data must be moved from disk to main memory foraccess, and written back for storage
Much slower accessthan main memory (more on this later)
Direct-access – possible to read data on disk in any order,unlike magnetic tapeHard disks vs. floppy disks Capacities range up to roughly 100 GB currently

Much larger capacity and cost/byte than main memory/ flash memory Growing constantly and rapidly with technology improvements
Survives power failures and system crashes

Disk failure candestroy data, but is very rare.

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 and DVD-R) Multiple write versions also available (CD-RW, DVDRW, and DVD-RAM) 4 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.

Optical Disks

Compact disk-read only memory (CD-ROM) Disks can be loaded into or removed from a drive 4 High storage capacity (640 MB per disk) High seek times or about 100 msec (optical read head is heavier and slower) Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks

Digital Video Disk (DVD)

DVD-5 holds 4.7 GB, and DVD-9 holds 8.5 GB DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB & 17 GB Other characteristics similar to CD-ROM 4 Record once versions (CD-R and DVD-R) Data can only be written once, and cannot be erased. High capacity and long lifetime; used for archival storage 4Multi-write versions (CD-RW, DVD-RW and DVDRAM) also available

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
4 Tape jukeboxes available for storing massive amounts of data Hundreds of terabytes (1 terabyte = 109 bytes) to even apetabyte (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

4E.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

Magnetic Hard Disk Mechanism

n 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 17,000 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. H 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

Performance Measures of Disks

Access Time – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:

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.) n **Data-Transfer Rate** – the rate at which data can be retrieved from or stored to the disk. H4 to 8 MB per second is typical

Multiple disks may share a controller, so rate that controller can handle is also important

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.

RAID

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 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)

o Techniques for using redundancy to avoid data loss are critical with large numbers of disks

 Originally a cost-effective alternative to large, expensive disks.

o I in RAID originally stood for ―inexpensive‘‘

o Today RAIDs are used for their higher reliability and bandwidth.

 The ―I‖ is interpreted as independent

 Improvement of Reliability viaRedundancy

 Redundancy– store extra information that can be used to rebuild information lost in a disk failure.

 E.g., Mirroring (or shadowing)

o Duplicate every disk. Logical disk consists of two physical disks.
o Every write is carried out on both disks
 Reads can take place from either disk
o 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.
Prob ability of combined event is very small o 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.
o 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
o Two main goals of parallelism in a disk system:
1. Load balance multiple small accesses to increase throughput
2. Parallelize large accesses to reduce response time.
o Improve transfer rate by striping data across multiple disks.
o 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
o 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

o 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
 RAID Level 0: Block striping; non-redundant.
    o Used in high-performance applications where data lost is not critical.
 RAID Level 1: Mirrored disks with block striping.
    o Offers best write performance.
    o Popular for applications such as storing log files in a database system.
 RAID Level 2: Memory-Style Error-Correcting-Codes (ECC) with bit striping.
  RAID Level 5: Block-Interleaved Distributed Parity; partitions data and parity NOTES amongall *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk.
o 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 4 disks. o Higher I/O rates than Level 4. Block writes occur in parallel if the blocks and their parity blocks are on different disks. o Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk. RAID Level 6: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
o Better reliability than Level 5 at a higher cost; not used as widely.

FILE OPERATIONS

 The database is stored as a collectionof *files*. Each file is a sequence of *records.* A record is a sequence of fields.

 One approach:

o assume record size is fixed.

o each file has records of one particular type only.

o different files are used for different relations.

 This case is easiestto implement; will consider variable length records later.

Fixed-Length Records

 Simple approach:

o Store record *i* starting from byte *n (i* – 1), where *n* is the size of each record.

o Record access is simple but records may cross blocks

 Modification: donot allow records to cross block boundaries.

  Free List s

o Store the address of the first deleted record in the file header.

o Use this first record to store the address of the second deleted record, and so on.

o Can think of these stored addresses as pointers since they ―point‖ to the location of a record.

o More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

Variable-Length Records

o 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 (used in some older data models).

o Byte string representation

 Attach an*end-of-record* ( ) control character to the end of each record.

 Difficulty with deletion.

 Difficulty with growth.

 Variable-Length Records: Slotted Page Structure

 Slotted page header contains: o number of record entries. o end of free space in the block. o 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 up-dated.

 Pointers should not point directly to record— instead they should point to the entry for the record in header.

 Fixed-length representation:

o reserved space

o 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

- 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 iswasted in all records except the first in a 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 clusteringNOTES file organization records of several different relations can be stored in the same file
- Motivation: store related records on the same block to minimize I/O

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
- Deletion– use pointer chains
- Insertion–locate the position where the record is to be inserted
- if there is free space insert there
- if no free space, insert the record in an overflow block
- In either case, pointer chain must be updated
- Need to reorganize thefile from time to time to restore sequential order

Clustering File Organization

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

Mapping of Objects to Files

- Mapping objects to files is similar to mapping tuples to files in a relational system; object data can be stored using file structures.
- Objects in O-O databases may lack uniformity and may be very large; such objects have to managed differently from records in a relational system.

o Set fields with a small number of elements may be implemented using data structures such as linked lists.
o Set fields with a larger number of elements may be implemented as separate relations in the database.
o Se t fields can also be eliminated at the storage level by normalization.

- Similar to conversion of multivalued attributes of E-R diagrams to relations
- Objects are identified by an object identifier (OID); the storage system needs a mechanism to

locate an object given its OID (this action is called dereferencing).
o logical identifiers do not directly specify an object‘s physical location; must maintain an index that
maps an OID to the object‘s actual location.
o physical identifiers encode the location of the object so the object can be found directly. Physical
OIDs typically have the following parts:
1. a volume or file identifier
2. a page identifier within the volume or file
3. an offset within the page

HASHING
Hashing is a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed.
Static Hashing
A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
Hash function is used to locate records for access, insertion as well as deletion.
Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization
Hash file organization of *account* file, using *branch-name* as key o There are 10 buckets ,
o The binary representation of the *i*th character is assumed to be the integer *i.*
o The hash function returns the sum of the binary representations of the characters modulo 10.
o E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3
o Hash file organization of *account* file, using *branch-name* as key

Hash Functions
o Worst had function maps all search-key values to the same bucket; this makes access time proportional to t he number of search-key values in the file.
o An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
o Ideal hash function is random, so each bucket will have the same number of records assigned to
it irrespective of the *actual distribution* of search-key values in the file.
o Typical hash functions perform computation on the internal binary representation of the searchkey.
o For example, for a string search-key, the binary representations of all the characters in the string
could be added and the sum modulo the number of buckets could be returned.

Handling of Bucket Overflows
o Bucket overflow can occur because of
 Insufficient buckets

 Skew in distribution of r ecords. This can occur due to two reasons:
 multiple records have same search-key value
 chosen hash function produces non-uniform distribution of key values

o Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled
by using *overflow buckets*.
o Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
o The Above scheme is called closed hashing.
o An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.

Hash Indices

o Hashing can be used not only for file organization, but also for index-structure creation.
o A hash index organizes the search keys, with their associated record pointers, into a hash file structure.
o Strictly speaking, hash indices are always secondary indices
o If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
o However, we use the term hash index to refer to both secondary index structures and hash organized files.
 Example of Hash Index

Deficiencies of Static Hashing

o In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
o Databases grow with time. If initial number of buckets is too small, performance will degrade due
to too much overflows.
o If file size at some point in the future is anticipated and number of buck-ets allocated accordingly,
significant amount of space will be wasted initially.
o If database shrinks, again space will be wasted.
o One option is periodic re-organization of the file with a new hash func-tion, but it is very expensive.
o These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

 Good for database that grows and shrinks in size .
 Allows the hash function to be modified dynamically.

Extendable hashing – one form of dynamic hashing

 Hash function generates values over a large range— typically *b*-bit integers, with $b = 32$.
 At any time use only a prefix of the hash function to index into a table of bucket addresses.
 Let the length of the prefix be *i* bits, $0 \ i \ 32$.

 Bucket address table size = 2i. Initially*i* = 0.
 Value of*i* grows and shrinks as the size of the database grows and shrinks.
 Multiple entries in the bucket address table may point to a bucket.
 Thus, actual number of buckets is < 2*i*.
 The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure
Use of Extendable Hash Structure
o Each bucket *j* stores a value *ij;* all the entries that point to the same bucket have the same values on the first *ij* bits.
o To locate the bucket containing search-key *Kj*: o 1. Compute *h(Kj) = X*
o 2.Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
o To insert a record with search-key value *Kj*
o follow same procedure as look-up and locate the bucket, say *j*.
o If there is room in the bucket *j* insert record in the bucket.
o Else the bucket must be split and insertion re-attempted.
o Overflow buckets used instead in some cases.

Updates in Extendable Hash Structure
o To split a bucket *j* when inserting record with search-key value *Kj*:
o If *i > ij* (more than one pointer to bucket *j*)
o allocate a new bucket *z*, and set *ij* and *iz* to the old *ij* -+ 1.
o make the second half of the bucket address table entries pointing to *j* to point to *z*
o remove and reinsert each record in bucket *j*.
o recompute new bucket for *Kj* and insert record in the bucket (fur-ther splitting is required if the bucket is still full)
o If *i = ij* (only one pointer to bucket *j*)
o increment *i* and double the size of the bucket address table.
o replace each entry in the table by two entries that point to the same bucket.
o recompute new bucket address table entry for *Kj* Now *i > ij* so use the first case above.
o When inserting a value, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further. o To delete a key value,
o locate it in its bucket and remove it.
o The bucket itself can be removed if it becomes empty (with appro-priate updates to the bucket address table).
o Coalescing of buckets can be done (can coalesce only with a ―buddy‖ bucket having same value of ij and same ij –1 prefix, if it is present).
o Decreasing bucket address table size is also possible.
o Note: decreasing bucket address table size is an expensive opera-tion and should be done only if number of buckets becomes much smaller than the size of the table.

Use of Extendable Hash Structure: Example Initial Hash structure, bucket size = 2
 Hash structure after insertion of one Brighton and two Downtown records
  Hash structure after insertion of Mianus record
 Hash structure after insertion of three Perryridge records
 Hash structure after insertion of Redwood and Round Hill records

Extendable Hashing vs. Other Schemes
  Benefits of extendable hashing :
o Hash performance does not degrade with growth of file
o Minimal space overhead
 Disadvantages of extendable hashing
o Extra level of indirection to find desired record
o Bucket address table may itself become very big (larger than memory)
 Need a tree structure to locate desired record in the structure!
o Changing size of bucket address table is an expensive operation
 Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost
of more bucket overflows.

INDEXING
 Indexing mechanisms used to speed up access to desired data. o E.g., author catalog in library
Search-key Pointer
 Search Key- attribute to set of attributes used to look up records in a file.
 An index file consists of records (called index entries) of the form.
 Index files are typically much smaller than the original file.
Two basic kinds of indices:
o Ordered indices: search keys are stored in sorted order.
o Hash indices: search keys are distributed uniformly across ―buckets‖ using a ―hash function‖. Index Evaluation Metrics
o Access types supported efficiently. E.g.,
o records with a specified value in the attribute
o or records with an attribute value falling in a specified range of values.
o Access time o Insertion time
o Deletion time
o Space overhead
 Ordered Indices
o Indexing techniques evaluated on basis of:
 In an ordered index, index entries are stored, sorted on the search key value. E.g., author catalogin library.
 Primary index: in a sequentially ordered file, the index whose search key specifies the sequentialorder of the file.

 Also called clustering index
 The search key of a primary index is usually but not necessarily the primary key.
 Secondary index: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
 Index-sequential file: ordered sequential file with
a primary index.
  Dense Index File s
o Dense index — Index record appears for every search key value in the file.

Sparse Index Files
 Sparse Index: contains index records for only some search-key values.
 Applicable when records are sequentially ordered on search-key
 To locate a record with search-key value *K* we:
 Find index record with largest search-key value < *K*
 Search file sequentially starting at the record to which the index record points.
 Less space and less maintenance overhead for insertions and deletions.
 Generally slower than dense index for locating records.
 Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.
 Example of Sparse Index Files

Multilevel Index
 If primary index does not fit in memory, access becomes expensive .
 To reducen umber of disk accesses to index records, treat primary
index kept on disk as a sequential file and construct a sparse index
on it. o outer index – a sparse index of primary index o inner index – the primary index file
  If even the outer index is too large to fit in main memory, yet another
level of index can be created, and so on.
  Indices at all levels must be updated on insertion or deletion from
the file.
 Index Update: Deletion
 If deleted record was the only record in the file with its particular search-key value, the searchkey
is deleted from the index also.
 Single-level index deletion:
o Dense indices – deletion of search-key is similar to file record deletion.
o Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
  Index Update: Insertion
o Single-level index insertion:
o Perform a lookup using the search-key value appearing
in the record to be inserted.
o Dense indices – if the search-key value does not appear
in the index, insert it.

o Sparse indices – if index stores an entry for each block of the file, no change needs to be made to
the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
o Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms.

Secondary Indices
o Frequently, one wants to find all the records whose
values in a certain field (which is not the search-key of the primary index satisfy some condition.
o Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch.
o Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
o We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
 Secondary Index on*balance* field of *account*
 Primary and Secondary Indice s
 Secondary indices have to be dense.
 Indices offer substantial benefits when searching for records.
 When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
 Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive.
 each record access may fetch a new block from disk
 Bitmap Indices
n Bitmap indices are a special type of index designed for efficient querying on multiple keys.
n Records in a relation are assumed to be numbered sequentially from, say,0
 Given a number n it must be easy to retrieve record n

 Particularly easy if records are of fixed size
n Applicable on attributes that take on a relatively small number of distinct values
 E.g. gender, country, state, …
 E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
n A bitmap is simply an array of bits.
n In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute.
 Bitmap has as many bits as records.
 In a bitmap for value v, the bit for a record is 1 if the record has the value v forthe attribute, and
is 0 otherwise.
n Bitmap indices are useful for queries on multiple attributes

 Not particularly useful for single attribute queries
n Queries are answered using bitmap operations
 Intersection (and)
 Union (or)
 Complementation (not)
n Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 Males with income level L1: 10010 AND 10100 = 10000

 If number of distinct attribute values is 8, bitmap is only 1% of relation size
n Deletion needs to be handled properly
 Existence bitmap to note if there is a valid record at a record location
 Needed for complementation
n Should keep bitmaps for all values, even null value.

B+-Tree Index Files
o B+-tree indices are an alternative to indexed-sequential files.
o Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
o Advantage of B+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance. o Disadvantage of B+-trees: extra insertion and deletion overhead,
space overhead.
o Advantages of B+-trees outweigh disadvantages, and they are used
extensively.
 A B+-tree is a rooted tree satisfying the following properties:
 All paths from root to leaf are of the same length
 Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
 A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
 Special cases:
If the root is not a leaf, it has at least 2 children.
If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.
 Typical node
o Ki are the search-key values
o Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
  The search-keys in a node are ordered
$K1 < K2 < K3 < . . . < Kn–1$.
Leaf Nodes in B+-Trees
o Properties of a leaf node:
  For i = 1, 2, . . ., $n–1$, pointer Pi either points to a file record with search-key value Ki, or to a bucket of pointers to file records, each record having search-key value Ki. Only need bucket structure if search-key does not form a primary key. If Li, Lj are leaf nodes

and□*i*□ <□*j*, *Li*‘s search-key values are less than *Lj*‘s search-key values.□ □*Pn*□ points to next leaf node in search-key order.

Non-Leaf Nodes in B+-Trees

□ Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:

o Al l the search-keys in the subtree to which *P*1 points are less than *K*1.

o For 2 *i n* – 1, all the search-keys in the subtree to which *Pi* points have values greater than or equal to *Ki*–1 and less than *Km–1*. □ □ Example of a B+-tree

B+-tree for *account* file (*n* = 3)

B+-tree for *account* file (*n* - 5)

o Leaf nodes must have between 2 and 4 v alues ((*n*–1)/2 and *n* –1, with *n* = 5).

o Non-leaf nodes other than root must have between 3 and 5 children ((*n*/2 and *n* with *n* =5).

o Root must have at least 2 children.

□ Observations about B+-trees

o Since the inter-node connections are done by pointers, ―logically‖ close blocks need not be ―physically‖ close.

o Th e non-leaf levels of the B+-tree form a hierarchy of sparse indices.

o The B+-tree contains a relatively small number of levels (logarithmic in the size of the main file),

thus searches can be conducted efficiently.

o Insertions and deletions to the main file can be handled efficiently, as the index can be restructured

in logarithmic time.

Queries on B+-Trees

□ Find all records with a search-key value of *k.* o Start with the root node

□ Examine the node for the smallest search-key value > *k.*

□ If such a value exists, assume it is*Kj.* Then follow *P*i to the child node.

□ Otherwise*k Km*–1, where there are *m* pointers in the node. Then follow *Pm* to the child node.

o If the node reached by following the pointer above is not a leaf node, repeat the above procedure

on the node, and follow the

corresponding pointer.

o Eventually reach a leaf node. If for some *i*, key *Ki* = k follow pointer *Pi* to the desired record or

bucket. Else no record with search-key value *k* exists.

Result of splitting node containing Brighton and Downtown on inserting NOTES

Clearview

B+-Tree before and after insertion of ―Clearview‖

□ Updates on B+-Trees: Deletion

□ Find the record to be deleted, and remove it from the main file and from the bucket (if present).

□ Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has

become empty.

□ If thenode has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then

 Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
 Delete the pair (*Ki*–1, *Pi)*, where *Pi* is the pointer to the deleted node, from its parent, recursively using the above procedure.
 Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
Update the corresponding search-key value in the parent of the node.

  The node deletions may cascade upwards till a node whichh as
n/2 or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the
root.
  Examples of B+-Tree Deletion
Before and after deleting ―Downtown‖
o The removal of the leaf node containing ―Downtown‖ did not result in its parent having too little
pointers. So the cascaded deletions stopped with the deleted leaf node‘s parent.

B+-Tree File Organization
o Index file degradation problem is solved by using B+-Tree indices. Data file degradation problem is solved by using B+-Tree File Organization.
o The leaf nodes in a B+-tree file organization store records, instead of pointers.
o Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
Leaf nodes are still required to be half full.
Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index
Example of B+-tree File Organization
o Good space utilization is important since records use more space than pointers.
o To improve space utilization, involve more sibling nodes in redistribution during splits and merges.
 Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node
having at least entries
Data Warehouse:
- Large organizations have complex internal organizations, and have data stored at different locations, on different operational (transaction processing) systems, under different schemas
- Data sources often store only current data, not historical data
- Corporate decision making requires a unified view of all organizational data, including historical data
- A data warehouse is a repository (archive) of information gathered from multiple

sources, stored under a unified schema, at a single site
- Greatly simplifies querying, permits study of historical trends
- Shifts decision support query load away from transaction processing systems

When and how to gather data
- Source driven architecture: data sources transmit new information to warehouse, either continuously or periodically (e.g. at night)
- Destination driven architecture: warehouse periodically requests new information from data sources
- Keeping warehouse exactly synchronized with data sources (e.g. using two-phase commit) is too expensive
- Usually OK to have slightly out-of-date data at warehouse
- Data/updates are periodically downloaded form online transaction processing (OLTP) systems. *What schema to use*
- Schema integration

Data cleansing
- E.g. correct mistakes in addresses
- E.g. misspellings, zip code errors

Merge address lists from different sources and purge duplicates
- Keep only one address record per household (―householding‖)

How to propagate updates
- Warehouse schema may be a (materialized) view of schema from data sources
- Efficient techniques for update of materialized views

What data to summarize
- Raw data may be too large to store on-line
- Aggregate values (totals/subtotals) often suffice
- Queries on raw data can often be transformed by query optimizer to use aggregate values.

Data Mining
- Broadly speaking, data mining is the process of semi-automatically analyzing large databases to find useful patterns.
- Like knowledge discovery in artificial intelligence data mining discovers statistical rules and patterns
- Differs from machine learning in that it deals with large volumes of data stored primarily on disk.
- Some types of knowledge discovered from a database can be represented by a set of rules. e.g.,: ―Young women with annual incomes greater than $50,000 are most likely to buy sports cars‖.
- Other types of knowledge represented by equations, or by prediction functions.
- Some manual intervention is usually required
- Pre-processing of data, choice of which type of pattern to find, postprocessing to find novel patterns

Applications of Data Mining
- **Prediction** based on past history
- Predict if a credit card applicant poses a good credit risk, based on some

attributes (income, job type, age, ..) and past history
- Predict if a customer is likely to switch brand loyalty
- Predict if a customer is likely to respond to ―junk mail‖
- Predict if a pattern of phone calling card usage is likely to be fraudulent

➢ Some examples of prediction mechanisms:

Mobile Databases

■ Recent advances in portable and wireless technology led to mobile computing, a new dimension in data communication and processing.

■ Portable computing devices coupled with wireless communications allow clients to access data from virtually anywhere and at any time.

■ There are a number of hardware and software problems that must be resolved before the capabilities of mobile computing can be fully utilized.

■ Some of the software problems – which may involve data management, transaction management, and database recovery – have their origins in distributed database systems.

■ In mobile computing, the problems are more difficult, mainly:

■ The limited and intermittent connectivity afforded by wireless communications.

■ The limited life of the power supply(battery).

■ The changing topology of the network.

■ In addition, mobile computing introduces new architectural possibilities and challenges.

Mobile Computing Architecture

■ **The general architecture of a mobile platform is illustrated in Fig 30.1.**

■ It is distributed architecture where a number of computers, generally referred to as Fixed Hosts and Base Stations are interconnected through a high-speed wired network.

■ Fixed hosts are general purpose computers configured to manage mobile units.

■ Base stations function as gateways to the fixed network for the Mobile Units.

■ **Wireless Communications** –

■ The wireless medium have bandwidth significantly lower than those of a wired network.

■ The current generation of wireless technology has data rates range from the tens to hundreds of kilobits per second (2G cellular telephony) to tens of megabits per second (wireless Ethernet, popularly known as WiFi).

■ Modern (wired) Ethernet, by comparison, provides data rates on the order of hundreds of megabits per second.

■ The other characteristics distinguish wireless connectivity options:

■ interference,

■ locality of access,

■ range,

■ support for packet switching,

■ seamless roaming throughout a geographical region.

■ Some wireless networks, such as WiFi and Bluetooth, use unlicensed areas of the frequency spectrum, which may cause interference with other appliances, such as cordless telephones.

■ Modern wireless networks can transfer data in units called packets, that are used

in wired networks in order to conserve bandwidth.

- **Client/Network Relationships –**
- Mobile units can move freely in a **geographic mobility domain**, an area that is circumscribed by wireless network coverage.
- To manage entire mobility domain is divided into one or more smaller domains, called **cells**, each of which is supported by at least one base station.
- Mobile units be unrestricted throughout the cells of domain, while maintaining information **access contiguity**.
- The communication architecture described earlier is designed to give the mobile unit the impression that it is attached to a fixed network, emulating a traditional client-server architecture.
- Wireless communications, however, make other architectures possible. One alternative is a mobile ad-hoc network (**MANET**), illustrated in 29.2.
- In a **MANET**, co-located mobile units do not need to communicate via a fixed network, but instead, form their own using cost-effective technologies such as Bluetooth.
- In a **MANET**, mobile units are responsible for routing their own data, effectively acting as base stations as well as clients.
- Moreover, they must be robust enough to handle changes in the network topology, such as the arrival or departure of other mobile units.
- MANET applications can be considered as peer-to-peer, meaning that a mobile unit is simultaneously a client and a server.
- Transaction processing and data consistency control become more difficult since there is no central control in this architecture.
- Resource discovery and data routing by mobile units make computing in a MANET even more complicated.
- Sample MANET applications are multi-user games, shared whiteboard, distributed calendars, and battle information sharing.

Characteristics of Mobile Environments

- The characteristics of mobile computing include:
- Communication latency
- Intermittent connectivity
- Limited battery life
- Changing client location
- The server may not be able to reach a client.
- A client may be unreachable because it is dozing – in an energy-conserving state in which many subsystems are shut down – or because it is out of range of a base station.
- In either case, neither client nor server can reach the other, and modifications must be made to the architecture in order to compensate for this case.
- Proxies for unreachable components are added to the architecture.
- For a client (and symmetrically for a server), the proxy can cache updates intended for the server.
- Mobile computing poses challenges for servers as well as clients.

- The latency involved in wireless communication makes scalability a problem.
- Since latency due to wireless communications increases the time to service each client request, the server can handle fewer clients.
- One way servers relieve this problem is by broadcasting data whenever possible.
- A server can simply broadcast data periodically.
- Broadcast also reduces the load on the server, as clients do not have to maintain active connections to it.
- Client mobility also poses many data management challenges.
- Servers must keep track of client locations in order to efficiently route messages to them.
- Client data should be stored in the network location that minimizes the traffic necessary to access it.
- The act of moving between cells must be transparent to the client.
- The server must be able to gracefully divert the shipment of data from one base to another, without the client noticing.
- Client mobility also allows new applications that are location-based.

Spatial Database

Types of Spatial Data

- **Point Data**
  - Points in a multidimensional space
  - E.g., Raster data such as satellite imagery, where each pixel stores a measured value
  - E.g., Feature vectors extracted from text
- **Region Data**
  - Objects have spatial extent with location and boundary.
  - DB typically uses geometric approximations constructed using line segments, polygons, etc., called vector data.

Types of Spatial Queries

- Spatial Range Queries
  - Find all cities within 50 miles of Madison
  - Query has associated region (location, boundary)
  - Answer includes ovelapping or contained data regions
- Nearest-Neighbor Queries
  - Find the 10 cities nearest to Madison
  - Results must be ordered by proximity
- Spatial Join Queries
  - Find all cities near a lake
  - Expensive, join condition involves regions and proximity

Applications of Spatial Data

- Geographic Information Systems (GIS)
  - E.g., ESRI‘s ArcInfo; OpenGIS Consortium
  - Geospatial information
  - All classes of spatial queries and data are common
- Computer-Aided Design/Manufacturing

 Store spatial objects such as surface of airplane fuselage
 Range queries and spatial join queries are common
- Multimedia Databases

 Images, video, text, etc. stored and retrieved by content
 First converted to feature vector form; high dimensionality
 Nearest-neighbor queries are the most common

Single-Dimensional Indexes

 B+ trees are fundamentally single-dimensional indexes.
 When we create a composite search key B+ tree, e.g., an index on <age, sal>, we effectively linearize the 2-dimensional space since we sort entries first by age and then by sal.

Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>

Multi-dimensional Indexes

 A multidimensional index clusters entries so as to exploit ―nearness‖ in multidimensional space.
 Keeping track of entries and maintaining a balanced index structure presents a challenge!

Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>

Motivation for Multidimensional Indexes

- Spatial queries (GIS, CAD).

 Find all hotels within a radius of 5 miles from the conference venue.
 Find the city with population 500,000 or more that is nearest to Kalamazoo, MI.
 Find all cities that lie on the Nile in Egypt.
 Find all parts that touch the fuselage (in a plane design).

- Similarity queries (content-based retrieval).

 Given a face, find the five most similar faces.

- Multidimensional range queries.

 50 < age < 55 AND 80K < sal < 90K

Drawbacks

- An index based on spatial location needed.

 One-dimensional indexes don‘t support multidimensional searching efficiently.
 Hash indexes only support point queries; want to support range queries as well.
 Must support inserts and deletes gracefully.

- Ideally, want to support non-point data as well (e.g., lines, shapes).
- The R-tree meets these requirements, and variants are widely used today.

Multimedia databases

- To provide such database functions as indexing and consistency, it is desirable to store multimedia data in a database

 Rather than storing them outside the database, in a file system

- The database must handle large object representation.
- Similarity-based retrieval must be provided by special index structures.
- Must provide guaranteed steady retrieval rates for continuous-media data.

Multimedia Data Formats

- Store and transmit multimedia data in compressed form
  - JPEG and GIF the most widely used formats for image data.
  - MPEG standard for video data use commonalties among a sequence of frames to achieve a greater degree of compression.
- MPEG-1 quality comparable to VHS video tape.
  - Stores a minute of 30-frame-per-second video and audio in approximately 12.5 MB
- MPEG-2 designed for digital broadcast systems and digital video disks; negligible loss of video quality.
  - Compresses 1 minute of audio-video to approximately 17 MB.
- Several alternatives of audio encoding
  - MPEG-1 Layer 3 (MP3), RealAudio, WindowsMedia format, etc.

Continuous-Media Data

- Most important types are video and audio data.
- Characterized by high data volumes and real-time information-delivery requirements.
  - Data must be delivered sufficiently fast that there are no gaps in the audio or video.
  - Data must be delivered at a rate that does not cause overflow of system buffers.
  - Synchronization among distinct data streams must be maintained
    - video of a person speaking must show lips moving synchronously with the audio

Video Servers

- **Video-on-demand** systems deliver video from central video servers, across a network, to terminals
  - must guarantee end-to-end delivery rates
- Current video-on-demand servers are based on file systems; existing database systems do not meet real-time response requirements.
- Multimedia data are stored on several disks (RAID configuration), or on tertiary storage for less frequently accessed data.
- Head-end terminals - used to view multimedia data
  - PCs or TVs attached to a small, inexpensive computer called a set-top box.

Similarity-Based Retrieval

Examples of similarity based retrieval

- Pictorial data: Two pictures or images that are slightly different as represented in the database may be considered the same by a user.
  - e.g., identify similar designs for registering a new trademark.
- Audio data: Speech-based user interfaces allow the user to give a command or identify a data item by speaking.
  - e.g., test user input against stored commands.
- Handwritten data: Identify a handwritten data item or command stored in the database