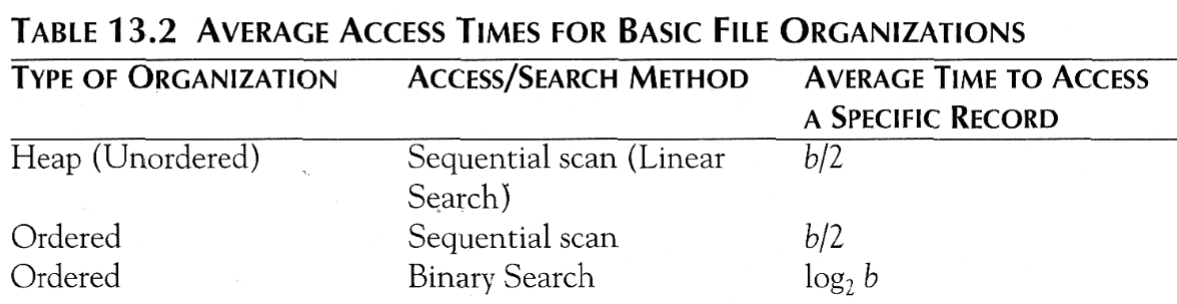
**Chapter 16:** ◼ Preferred secondary storage device for high storage capacity and low cost. ◼ Data stored as magnetized areas on magnetic disk surfaces. ◼ A **disk pack** contains several magnetic disks connected to a rotating spindle. ◼ Disks are divided into concentric circular **tracks** on each disk **surface**. ◼ Track capacities vary typically from 4 to 50 Kbytes or more. ◼ A track is divided into smaller **blocks** or **sectors** ◼ because it usually contains a large amount of information ◼ The division of a track into sectors is hard-coded on the disk surface and cannot be changed. ◼ One type of sector organization calls a portion of a track that subtends a fixed angle at the center as a sector. ◼ A track is divided into **blocks**. ◼ The block size B is fixed for each system. ◼ Typical block sizes range from B=512 bytes to B=4096 bytes. ◼ Whole blocks are transferred between disk and main memory for processing. ◼ **A read-write head** moves to the track that contains the block to be transferred. ◼ Disk rotation moves the block under the read-write head for reading or writing. ◼ A physical disk block (hardware) address consists of: **◼ a cylinder number** (imaginary collection of tracks of same radius from all recorded surfaces) ◼ **the track number** or surface number (within the cylinder) ◼ **and block number** (within track). ◼ Reading or writing a disk block is time consuming because of the seek time s and rotational delay (latency) rd. ◼ Double buffering can be used to speed up the transfer of contiguous disk blocks. ◼ Fixed and variable length records ◼ Records contain fields which have values of a particular type ◼ E.g., amount, date, time, age ◼ Fields themselves may be fixed length or variable length ◼ Variable length fields can be mixed into one record: ◼ Separator characters or length fields are needed so that the record can be “parsed.” ◼ **Blocking**: ◼ Refers to storing a number of records in one block on the disk. ◼ **Blocking factor** (bfr) refers to the number of records per block. ◼ There may be empty space in a block if an integral number of records do not fit in one block. ◼ **Spanned Records**: ◼ Refers to records that exceed the size of one or more blocks and hence span a number of blocks. ◼ A **file** is a sequence of records, where each record is a collection of data values (or data items). ◼ A **file descriptor** (or **file header**) includes information that describes the file, such as the field names and their data types, and the addresses of the file blocks on disk. ◼ Records are stored on disk blocks. ◼ The **blocking factor bfr for a file** is the (average) number of file records stored in a disk block. ◼ A file can have **fixed-length** records or **variable-length** records. ◼ File records can be unspanned or spanned ◼ **Unspanned**: no record can span two blocks ◼ **Spanned**: a record can be stored in more than one block ◼ The physical disk blocks that are allocated to hold the records of a file can be contiguous, linked, or indexed. ◼ In a file of fixed-length records, all records have the same format. Usually, unspanned blocking is used with such files. ◼ Files of variable-length records require additional information to be stored in each record, such as **separator characters** and **field types**. ◼ Usually spanned blocking is used with such files. ◼ Typical file operations include: ◼ **OPEN**: Readies the file for access, and associates a pointer that will refer to a current file record at each point in time. ◼ **FIND**: Searches for the first file record that satisfies a certain condition, and makes it the current file record. ◼ **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record. ◼ **READ**: Reads the current file record into a program variable. **◼ INSERT**: Inserts a new record into the file & makes it the current file record. ◼ **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid. ◼ **MODIFY**: Changes the values of some fields of the current file record. ◼ **CLOSE**: Terminates access to the file. ◼ **REORGANIZE**: Reorganizes the file records. ◼ For example, the records marked deleted are physically removed from the file or a new organization of the file records is created. ◼ **READ\_ORDERED**: Read the file blocks in order of a specific field of the file. **Unordered Files**: ◼ Also called a heap or a pile file. ◼ New records are inserted at the end of the file. ◼ A linear search through the file records is necessary to search for a record. ◼ This requires reading and searching half the file blocks on the average, and is hence quite expensive. ◼ Record insertion is quite efficient. ◼ Reading the records in order of a particular field requires sorting the file records. **Ordered Files**: ◼ Also called a sequential file. ◼ File records are kept sorted by the values of an ordering field. ◼ Insertion is expensive: records must be inserted in the correct order. ◼ It is common to keep a separate unordered overflow (or transaction) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file. ◼ A binary search can be used to search for a record on its ordering field value. ◼ This requires reading and searching log2 of the file blocks on the average, an improvement over linear search. ◼ Reading the records in order of the ordering field is quite efficient. **Hashed Files**: ◼ Hashing for disk files is called External Hashing ◼ The file blocks are divided into M equal-sized buckets, numbered bucket0, bucket1, ..., bucketM-1. ◼ Typically, a bucket corresponds to one (or a fixed number of) disk block. ◼ One of the file fields is designated to be the hash key of the file. ◼ The record with hash key value K is stored in bucket i, where i=h(K), and h is the hashing function. ◼ Search is very efficient on the hash key. ◼ Collisions occur when a new record hashes to a bucket that is already full. ◼ An overflow file is kept for storing such records. ◼ Overflow records that hash to each bucket can be linked together. ◼ There are numerous methods for collision resolution, including the following: ◼ **Open addressing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found. ◼ **Chaining**: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. ◼ **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary. ◼ To reduce overflow records, a hash file is typically kept 70-80% full. ◼ The hash function h should distribute the records uniformly among the buckets ◼ Otherwise, search time will be increased because many overflow records will exist. ◼ **Main disadvantages of static external hashing**: ◼ Fixed number of buckets M is a problem if the number of records in the file grows or shrinks. ◼ Ordered access on the hash key is quite inefficient (requires sorting the records). ◼ **Dynamic and Extendible Hashing Techniques** ◼ Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records. ◼ These techniques include the following: **dynamic hashing, extendible hashing, and linear hashing**. ◼ Both dynamic and extendible hashing use the binary representation of the hash value h(K) in order to access a directory. ◼ In dynamic hashing the directory is a binary tree. ◼ In extendible hashing the directory is an array of size 2d where d is called the global depth. ◼ The directories can be stored on disk, and they expand or shrink dynamically. ◼ Directory entries point to the disk blocks that contain the stored records. ◼ An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks. ◼ The directory is updated appropriately. ◼ Dynamic and extendible hashing do not require an overflow area. ◼ Linear hashing does require an overflow area but does not use a directory. ◼ Blocks are split in ***linear*** order as the file expands. **Parallelizing Disk Access using RAID Technology.** ◼ Secondary storage technology must take steps to keep up in performance and reliability with processor technology. ◼ A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**. ◼ The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors. ◼ A natural solution is a large array of small independent disks acting as a single higher-performance logical disk. ◼ A concept called **data striping** is used, which utilizes parallelism to improve disk performance. ◼ Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk. ◼ Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. ◼ Raid level 0 has no redundant data and hence has the best write performance at the risk of data loss ◼ **Raid level 1** uses mirrored disks. ◼ Raid level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. **Level 2** includes both error detection and correction. ◼ **Raid level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed. ◼ **Raid Levels 4** and 5 use block-level data striping, with **level 5** distributing data and parity information across all disks. ◼ **Raid level 6** applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks. **Use of RAID**◼ Different raid organizations are being used under different situations ◼ **Raid level 1** (mirrored disks) is the easiest for rebuild of a disk from other disks ◼ It is used for critical applications like logs **◼ Raid level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. ◼ Level 2 includes both error detection and correction. ◼ **Raid level 3** (single parity disks relying on the disk controller to figure out which disk has failed) and **level 5** (block-level data striping) are preferred for Large volume storage, with level 3 giving higher transfer rates. ◼ Most popular uses of the RAID technology currently are: ◼ **Level 0** (with striping), **Level 1** (with mirroring) and **Level 5** with an extra drive for parity. ◼ Design Decisions for RAID include: ◼ Level of RAID, number of disks, choice of parity schemes, and grouping of disks for block-level striping. **Storage Area Networks**: ◼ The demand for higher storage has risen considerably in recent times. ◼ Organizations have a need to move from a static fixed data center oriented operation to a more flexible and dynamic infrastructure for information processing. ◼ Thus they are moving to a concept of Storage Area Networks (SANs). ◼ In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner. ◼ This allows storage systems to be placed at longer distances from the servers and provide different performance and connectivity options. ◼ **Advantages of SANs** are: ◼ Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches. ◼ Up to 10km separation between a server and a storage system using appropriate fiber optic cables. ◼ Better isolation capabilities allowing non-disruptive addition of new peripherals and servers. ◼ SANs face the problem of combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware.



**Types of Single-level Ordered Indexes:** - Primary Indexes - Clustering Indexes - Secondary Indexes: A **single-level index** is an auxiliary file that makes it more efficient to search for a record in the data file. ◼ The index is usually specified on one field of the file (although it could be specified on several fields) ◼ One form of an index is a file of entries <field value, pointer to record>, which is ordered by field value ◼ The index is called an access path on the field.The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller ◼ A binary search on the index yields a pointer to the file record ◼ A dense index has an index entry for every search key value (and hence every record) in the data file. ◼ A sparse (or nondense) index, on the other hand, has index entries for only some of the search values.

**record size R=150 bytes, block size B=512 bytes, r=30000 records, blocking factor Bfr= └B/R┘ = └ 512/150 ┘ = 3 records/block ◼ number of file blocks b= ┌ r/Bfr ┐ = ┌ 30000/3 ┐ = 10000 blocks.** For an index on the SSN field, assume the field size VSSN=9 bytes, assume the record pointer size PR=7 bytes. Then: ◼ index entry size RI=(VSSN+ PR)=(9+7)=16 bytes ◼ index blocking factor BfrI= └B/RI┘ = └ 512 /16 ┘ = 32 entries/block ◼ number of index blocks bI= ┌ r/ BfrI ┐ = ┌ 30000/32 ┐ = 938 blocks ◼ binary search needs log2bI= log2938= 10 block accesses ◼ This is compared to an average linear search cost of: ◼ ┌ b/2┐ = 10000/2= 5000 block accesses ◼ If the file records are ordered, the binary search cost would be: ◼ ┌ log2b ┐ = log210000= 14 block accesses

**Primary Index** ◼ Defined on an ordered data file ◼ The data file is ordered on a key field ◼ Includes one index entry for each block in the data file; the index entry has the key field value for the first record in the block, which is called the block anchor ◼ A similar scheme can use the last record in a block. ◼ A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

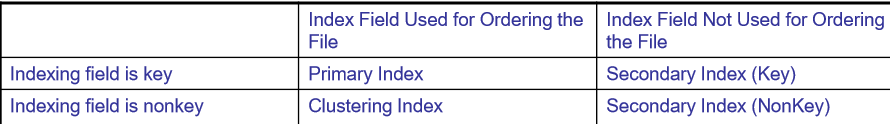
**Eg 1:** Suppose that: ◼ record size R=100 bytes block size B=1024 bytes r=30000 records ◼ file records are fixed and unspanned ◼ Then, we get: ◼ blocking factor Bfr= └B/R┘= └1024/100┘ = 10 records/block ◼ number of file blocks b= ┌r/Bfr ┐= (30000/10)= 3000 blocks ◼ binary search on data file ◼ ┌ log2b ┐= ┌ log23000 ┐= 12 block accesses ◼ For an index on the SSN field, assume the field size VSSN=9 bytes, assume the record pointer size PR=6 bytes. Then: ◼ index entry size RI=(VSSN+ PR)=(9+6)=15 bytes ◼ index blocking factor BfrI= └ B/RI ┘= └1024/15┘ = 68 entries/block ◼ the total number of index entries rI=3000 (why?) ◼ number of index blocks bI= ┌rI/ BfrI ┐= ┌ 3000/68 ┐= 45 blocks ◼ binary search needs ┌ log2bI ┐= ┌ log245 ┐ = 6 block accesses ◼ Total we need 7 = 6 + 1 block access (1 for data file)

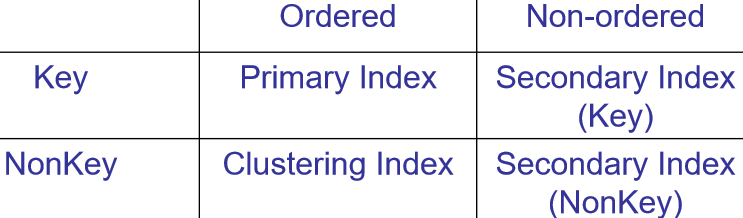
**Clustering Index** ◼ Defined on an ordered data file ◼ The data file is ordered on a non-key field unlike primary index, which requires that the ordering field of the data file have a distinct value for each record. ◼ Includes one index entry for each distinct value of the field; the index entry points to the first data block that contains records with that field value. ◼ It is another example of nondense index where Insertion and Deletion is relatively straightforward with a clustering index. A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file and also with a separate block cluster for each group of records that share the same value for the clustering field.

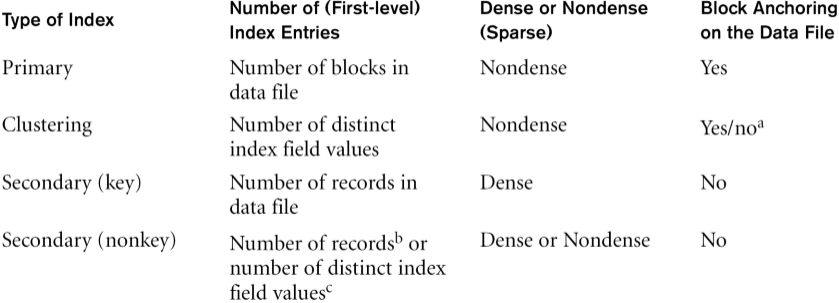
**Secondary Index** ◼ A secondary index provides a secondary means of accessing a file for which some primary access already exists. ◼ The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values. ◼ The index is an ordered file with two fields. ◼ The first field is of the same data type as some non-ordering field of the data file that is an indexing field. ◼ The second field is either a block pointer or a record pointer. ◼ There can be many secondary indexes (and hence, indexing fields) for the same file. ◼ Includes one entry for each record in the data file; hence, it is a dense index. A dense secondary index (with block pointers) on a nonordering key field of a file.

**Eg 2:** r=30,000 fixed-length records, R=100 bytes, B=1,024 bytes, and b= 3000 blocks. ◼ Linear search: b/2 = 3000/2 = 1500 block accesses ◼ Secondary index on a nonordering key field: V=9 bytes, and P=6 bytes ◼ Ri = (9+6) = 15 bytes ◼ bfri =  B/Ri  =  1024/15  = 68 entries/block (floor) ◼ # of index entries, ri = r since dense ◼ bi =  ri/bfri =  30,000/68  = 442 blocks (ceiling) ◼ Binary search needs  log2bi =  log2442 = 9 block accesses (ceiling) ◼ Total block accesses = 9 + 1 = 10

**For nonkey, non-ordering field:** Option 1: ◼ Duplicate index entries with the same K(i) values. Dense index ◼ Option 2: ◼ Variable length records for the index entries (repeating pointer): e.g. <P(i,1), …, P(i,k)> for K(i) ◼ Option 3: ◼ Create extra level to handle the multiple pointers







**Multi-Level Indexes** ◼ a single-level index is an ordered file, we can create a primary index to the index itself; ◼ original index file is first-level index & index to the index is second-level index. ◼ We can repeat until all entries of the top level fit in one disk block ◼ can be created for any type of firstlevel index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block

**Eg 3: Convert Example 2 into a multilevel index** ◼ bfri = fo (fan-out) = 68 ◼ # of first-level blocks b1 = 442 blocks ◼ # of second-level blocks b2 = b1/fo = 442/68  =7 blocks (ceiling) ◼ # of third-level blocks b3 = b2/fo = 7.68 = 1 block (ceiling)◼ Therefore, third level is top level (t=3) ◼ Total block accesses = t+1 = 4 block accesses (all ceiling here)

Such a multi-level index is a form of search tree ◼ However, insertion and deletion of new index entries is a severe problem because every level of the index is an ordered file. ◼ Dynamic multilevel index: leaves some space in each of its block for inserting new entries ◼ That is called B-tree or B+-tree

**Dynamic Multilevel Indexes:** ◼ Tree data structure ◼ A tree is formed of nodes ◼ Each node has one parent node (except root) and several child nodes. ◼ A root does not have parent node ◼ A leaf does not have child node ◼ A subtree of a node consists of that node and all its descendant nodes

**Search Tree:** ◼ A search tree of order p is a tree such that ◼ Each node contains at most p-1 search values, and ◼ P pointers in the order of <P1, K1, P2, K2,…, Pq-1, Kq-1, Pq> (Pi is pointer to a child node, and Ki is a search value) ◼ Two constraints must hold at all times on the search tree 1. Within each node K1 < K2 < … < Kq-1 2. For all values X in the subtree pointed at by P, we have Ki-1 <X< Ki for 1<i<q; X< Ki for i=1, and Ki-1<X for i=q

**Dynamic Multilevel Indexes Using B-Trees and B+-Trees:** ◼ Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem ◼ This leaves space in each tree node (disk block) to allow for new index entries ◼ These data structures are variations of search trees that allow efficient insertion and deletion of new search values. ◼ In B-Tree and B+-Tree data structures, each node corresponds to a disk block ◼ Each node is kept between half-full and completely full. ◼ An insertion into a node that is not full is quite efficient ◼ If a node is full the insertion causes a split into two nodes ◼ Splitting may propagate to other tree levels ◼ A deletion is quite efficient if a node does not become less than half full ◼ If a deletion causes a node to become less than half full, it must be merged with neighboring nodes.

**Difference between B-tree and B+-tree:** ◼ In a B-tree, pointers to data records exist at all levels of the tree ◼ In a B+-tree, all pointers to data records exists at the leaf-level nodes ◼ A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree.

**B-tree of order p:** 1. Each internal node in the B-tree <P1,<K1,Pr1>,P2,<K2,Pr2>, …, <Kq-1,Prq-1>,Pq> where q  p Pi: tree pointer, and Pri: data pointer 2. Within each node, K1<K2<. . .<Kq-1 3. For all search key field values X in the subtree Ki-1<X<Ki for 1<i<q, X<Ki for i=1, and Ki-1<X for i=q 4. Each node has at most p tree pointers 5. Each node has at least p/2 tree pointers 6. A node with q tree pointers, q  p, has q-1 search key field values 7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes

**Order p of B-tree:** ◼ Search field V=9 bytes, ◼ Block size B=512 bytes, ◼ Data pointer Pr=7 bytes, and ◼ Block pointer P=6 bytes ◼ At most p tree pointers, p-1 data pointers, and p-1 search key fields, which should be in a single block ◼ (p\*P) + ((p-1)\*(Pr+V)) <= B; (p\*6) + ((p-1)\*(7+9)) <= 512; (22\*p) <= 528 Therefore, p = 23 (not 24).

**#of Blocks and Levels:** ◼ Search field of Example 4 = nonordering key field ◼ Each node of B-tree is 69 % full ◼ Each node will have p\*0.69 = 23 \* 0.69 = 16 pointers, and 15 search key field values. fo = 16 ◼ Root: 1 node 15 entries 16pointers Level1: 16 nodes 240 entries 256 pointers Level2: 256 nodes 3840 entries 4096 pointers Level3: 4096 nodes 61440 entries ◼ For example, two-level (3840+240+15=4095), or three-level (65,535 entries).

**B+-tree of order p (Internal nodes):** 1. Each internal node is <P1,K1,P2,K2, …,Pq-1,Kq-1 Pq> where q  p, each Pi is a tree pointer 2. Within each internal node, K1<K2<. . .<Kq-1 3. For all search field values X in the subtree pointed at by Pi, Ki-1<X<Ki for 1<i<q, X<Ki for i=1, and Ki-1<X for i=q 4. Each internal node has at most p tree pointers 5. Each internal node has at least p/2 tree pointers 6. An internal node with q pointers q  p, has q-1 search field values

**B+-tree of order p (leaf nodes):** 1. Each leaf node is <<K1,Pr1>, <K2,Pr2>,…, <Kq-1,Prq-1>,Pnext> where q  p, each Pri is a data pointer, and Pnext points to the next leaf node 2. Within each leaf node, K1<K2<. . .<Kq-1,q  p 3. Each Pri is a data pointer points to Ki search field value 4. Each leaf node has at least p/2 tree pointers 5. All leaf nodes are at the same level

**Order p of B+-tree:** ◼ Search field V=9 bytes, ◼ Block size B=512 bytes, ◼ Data pointer Pr=7 bytes, and ◼ Block pointer P=6 bytes ◼ At most p tree pointers, and p-1 search key fields, which should be in a single block ◼ (p\*P) + ((p-1)\*V)  B (p\*6) + ((p-1)\*9)  512 (15\*p)  521 Therefore, p = 34 ◼ The leaf node order pleaf (pleaf\*(Pr+V)) + P  B (pleaf\*(7+9) + 6)  512 (16\* pleaf)  506 ➔ pleaf = 31

**# of Entries:** ◼ Construct B+-tree for Example 6 ◼ Each node of B-tree is 69 % full ◼ Each node will have 34 \* 0.69 = 23 pointers, and 22 search key field values. ◼ Each leaf node 0.69\*pleaf = 0.69 \* 31 ➔ 21 data record pointer ◼ Root: 1 node 22 entries 23 pointers ◼ Level1: 23 nodes 506 entries 529 pointers ◼ Level2: 529 nodes 11,638 entries 12,167 pointers ◼ Level3: 12,167 nodes 255,507 data record pointer ◼ For example, three-level B+-tree 255,507 record pointers.

Chapter 14:

**Informal Design Guidelines - relational database design** - grouping of attributes to form "good" relation schemas. Two levels of relation schemas ◼ The logical "user view" level ◼ The storage "base relation" level. **Formal concepts of functional dependencies and normal forms** ◼ - 1NF (First Normal Form) ◼ - 2NF (Second Normal Form) ◼ - 3NF (Third Noferferferfewrmal Form) ◼ - BCNF (Boyce-Codd Normal Form).

**Semantics of the Relational Attributes must be clear. GUIDELINE 1:** Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes). ◼ Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation ◼ Only foreign keys should be used to refer to other entities ◼ Entity and relationship attributes should be kept apart as much as possible. ◼ Bottom Line: Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

**Redundant Information in Tuples and Update Anomalies:** Information is stored redundantly ◼ Wastes storage ◼ Causes problems with update anomalies ◼ Insertion anomalies ◼ Deletion anomalies ◼ Modification anomalies

**Eg OF AN UPDATE ANOMALY:** ◼ Consider the relation: ◼ EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours) ◼ Update Anomaly: ◼ Changing the name of project number P1 from “Billing” to “Customer-Accounting” may cause this update to be made for all 100 employees working on project P1.

**EXAMPLE OF AN INSERT ANOMALY:** ◼ Consider the relation: ◼ EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours) ◼ Insert Anomaly: ◼ Cannot insert a project unless an employee is assigned to it. ◼ Conversely ◼ Cannot insert an employee unless an he/she is assigned to a project.

**EXAMPLE OF A DELETE ANOMALY:** ◼ Consider the relation: ◼ EMP\_PROJ(Emp#, Proj#, Ename, Pname, No\_hours) ◼ Delete Anomaly: ◼ When a project is deleted, it will result in deleting all the employees who work on that project. ◼ Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

**Guideline for Redundant Information in Tuples and Update Anomalies: ◼ GUIDELINE 2:** ◼ Design a schema that does not suffer from the insertion, deletion and update anomalies. ◼ If there are any anomalies present, then note them so that applications can be made to take them into account.

**Null Values in Tuples ◼ GUIDELINE 3:** ◼ Relations should be designed such that their tuples will have as few NULL values as possible ◼ Attributes that are NULL frequently could be placed in separate relations (with the primary key) ◼ Reasons for nulls: ◼ Attribute not applicable or invalid ◼ Attribute value unknown (may exist) ◼ Value known to exist, but unavailable.

**Generation of Spurious Tuples – avoid at any cost:** ◼ Bad designs for a relational database may result in erroneous results for certain JOIN operations ◼ The "lossless join" property is used to guarantee meaningful results for join operations . ◼ GUIDELINE 4: ◼ The relations should be designed to satisfy the lossless join condition. ◼ No spurious tuples should be generated by doing a natural-join of any relations. **Spurious Tuples (2)** ◼ There are two important properties of decompositions: a) Non-additive or losslessness of the corresponding join b) Preservation of the functional dependencies. ◼ Note that: ◼ Property (a) is extremely important and cannot be sacrificed. ◼ Property (b) is less stringent and may be sacrificed.

**Functional Dependencies** ◼ Functional dependencies (FDs) ◼ Are used to specify formal measures of the "goodness" of relational designs ◼ And keys are used to define normal forms for relations ◼ Are constraints that are derived from the meaning and interrelationships of the data attributes ◼ A set of attributes X functionally determines a set of attributes Y if the value of X determines a unique value for Y. **Defining Functional Dependencies** ◼ X  Y holds if whenever two tuples have the same value for X, they must have the same value for Y ◼ For any two tuples t1 and t2 in any relation instance r(R): If t1[X]=t2[X], then t1[Y]=t2[Y] ◼ X  Y in R specifies a constraint on all relation instances r(R) ◼ Written as X  Y; can be displayed graphically on a relation schema as in Figures. ( denoted by the arrow: ). ◼ FDs are derived from the real-world constraints on the attributes. **Examples of FD constraints (1)** ◼ Social security number determines employee name ◼ SSN  ENAME ◼ Project number determines project name and location ◼ PNUMBER  {PNAME, PLOCATION} ◼ Employee ssn and project number determines the hours per week that the employee works on the project ◼ {SSN, PNUMBER}  HOURS. **Examples of FD constraints (2)** ◼ An FD is a property of the attributes in the schema R ◼ The constraint must hold on every relation instance r(R) ◼ If K is a key of R, then K functionally determines all attributes in R ◼ (since we never have two distinct tuples with t1[K]=t2[K]) . **Defining FDs from instances** ◼ Note that in order to define the FDs, we need to understand the meaning of the attributes involved and the relationship between them. ◼ An FD is a property of the attributes in the schema R ◼ Given the instance (population) of a relation, all we can conclude is that an FD may exist between certain attributes. ◼ What we can definitely conclude is – that certain FDs do not exist because there are tuples that show a violation of those dependencies. **Ruling Out FDs** Note that given the state of the TEACH relation, we can say that the FD: Text → Course may exist. However, the FDs Teacher → Course, Teacher → Text and Couse → Text are ruled out.

Normalization of Relations: ◼ Normalization: ◼ The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations. ◼ Normal form: ◼ Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form. ◼ 2NF, 3NF, BCNF ◼ based on keys and FDs of a relation schema ◼ 4NF ◼ based on keys, multi-valued dependencies : MVDs; ◼ 5NF ◼ based on keys, join dependencies : JDs ◼ Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation.

Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties ◼ The practical utility of these normal forms becomes questionable when the constraints on which they are based are hard to understand or to detect ◼ The database designers need not normalize to the highest possible normal form ◼ (usually up to 3NF and BCNF. 4NF rarely used in practice.) ◼ Denormalization: ◼ The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form. A superkey of a relation schema R = {A1, A2, ...., An} is a set of attributes S subset-of R with the property that no two tuples t1 and t2 in any legal relation state r of R will have t1[S] = t2[S] ◼ A key K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more. If a relation schema has more than one key, each is called a candidate key. ◼ One of the candidate keys is arbitrarily designated to be the primary key, and the others are called secondary keys. ◼ A Prime attribute must be a member of some candidate key ◼ A Nonprime attribute is not a prime attribute— that is, it is not a member of any candidate key.

**First Normal Form:** ◼ Disallows ◼ composite attributes ◼ multivalued attributes ◼ nested relations; attributes whose values for an individual tuple are non-atomic ◼ Considered to be part of the definition of a relation ◼ Most RDBMSs allow only those relations to be defined that are in First Normal Form. **Second Normal Form:** ◼ Uses the concepts of FDs, primary key ◼ Definitions ◼ Prime attribute: An attribute that is member of the primary key K ◼ Full functional dependency: a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more ◼ Examples: ◼ {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold ◼ {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency ) since SSN -> ENAME also holds. A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on the primary key ◼ R can be decomposed into 2NF relations via the process of 2NF normalization or “second normalization”

Third Normal Form: ◼ Definition: ◼ Transitive functional dependency: a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z ◼ Examples: ◼ SSN -> DMGRSSN is a transitive FD ◼ Since SSN -> DNUMBER and DNUMBER -> DMGRSSN hold ◼ SSN -> ENAME is non-transitive ◼ Since there is no set of attributes X where SSN -> X and X -> ENAME. ◼ A relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key ◼ R can be decomposed into 3NF relations via the process of 3NF normalization ◼ NOTE: ◼ In X -> Y and Y -> Z, with X as the primary key, we consider this a problem only if Y is not a candidate key. ◼ When Y is a candidate key, there is no problem with the transitive dependency . ◼ E.g., Consider EMP (SSN, Emp#, Salary ). ◼ Here, SSN -> Emp# -> Salary and Emp# is a candidate key.

Normal Forms Defined Informally ◼ 1st normal form ◼ All attributes depend on the key ◼ 2nd normal form ◼ All attributes depend on the whole key ◼ 3rd normal form ◼ All attributes depend on nothing but the key. **General Normal Form Definitions (For Multiple Keys) -** Any attribute involved in a candidate key is a prime attribute ◼ All other attributes are called non-prime attributes.

**General Definition of 2NF (For Multiple Candidate Keys) -** A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on every key of R ◼ In Figure 14.12 the FD County\_name → Tax\_rate violates 2NF. So second normalization converts LOTS into LOTS1 (Property\_id#, County\_name, Lot#, Area, Price) LOTS2 ( County\_name, Tax\_rate).

**General Definition of Third Normal Form: ◼** Definition: ◼ Superkey of relation schema R - a set of attributes S of R that contains a key of R ◼ A relation schema R is in third normal form (3NF) if whenever a FD X → A holds in R, then either: ◼ (a) X is a superkey of R, or ◼ (b) A is a prime attribute of R ◼ LOTS1 relation violates 3NF because Area → Price ; and Area is not a superkey in LOTS1.

**Interpreting the General Definition of Third Normal Form** ◼ Consider the 2 conditions in the Definition of 3NF: A relation schema R is in third normal form (3NF) if whenever a FD X → A holds in R, then either: ◼ (a) X is a superkey of R, or ◼ (b) A is a prime attribute of R ◼ Condition (a) catches two types of violations : - one where a prime attribute functionally determines a non-prime attribute. This catches 2NF violations due to non-full functional dependencies. -second, where a non-prime attribute functionally determines a non-prime attribute. This catches 3NF violations due to a transitive dependency.

**Interpreting the General Definition of Third Normal Form:** ◼ ALTERNATIVE DEFINITION of 3NF: We can restate the definition as: A relation schema R is in third normal form (3NF) if every non-prime attribute in R meets both of these conditions: ◼ It is fully functionally dependent on every key of R ◼ It is non-transitively dependent on every key of R Note that stated this way, a relation in 3NF also meets the requirements for 2NF. ◼ The condition (b) from the last slide takes care of the dependencies that “slip through” (are allowable to) 3NF but are “caught by” BCNF which we discuss next.