Johns Hopkins Engineering

Principles of Database Systems

Module 11 / Lecture 1

File Structures and Indexes

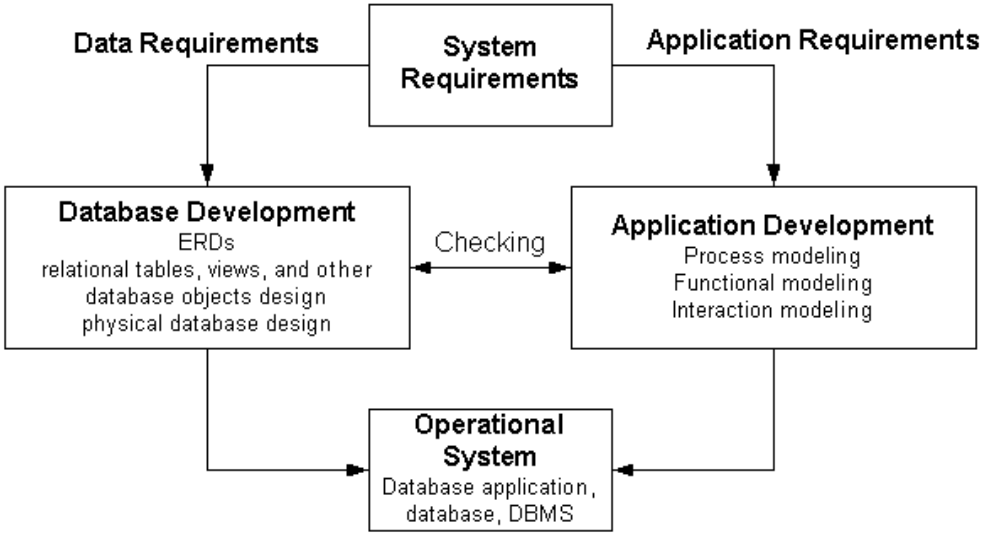
JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Enterprise Architecture

- Business Architecture:
  - Operating Scenarios
  - Business Process Model, Data Flow Diagram, UML Use Case
- Information Architecture:
  - Conceptual, Logical, and Physical Models, UML Class diagram, XML model
- Application Architecture:
  - Functional Model, System/Interface Model, Reuse/Collaboration Models
- Technology Architecture:
  - OS, HW, DBMS, Standards, Network, Communications, Security

Database and Database Application Development

- Paths for Database and Database Application Development



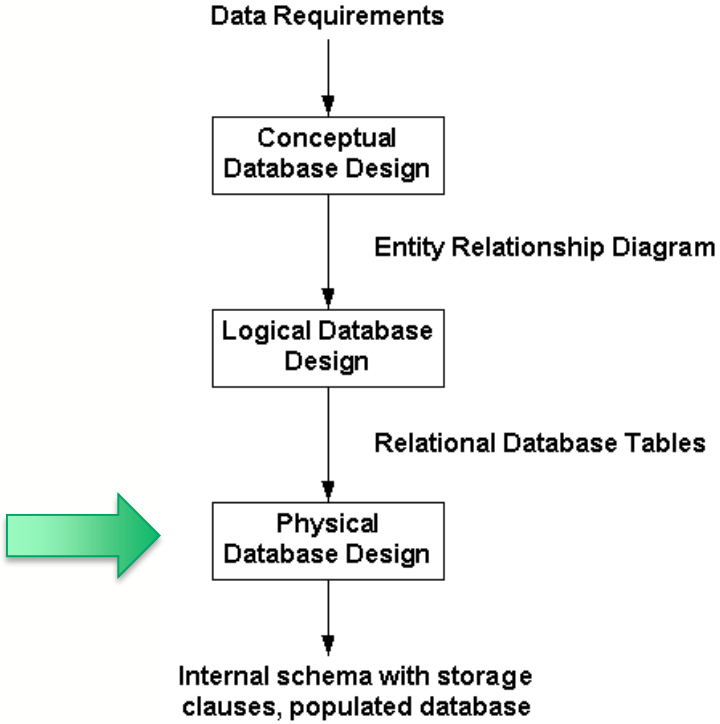
Interaction between database and application development

Business Rules

- Structural assertion
  - DDLs with constraints and domains
- Derivation
  - GPA, line total, grand total
- Action assertion
  - Decision logic – type of services/members, course registration, task assignments based on rules and roles
  - Calculation formula – sales tax

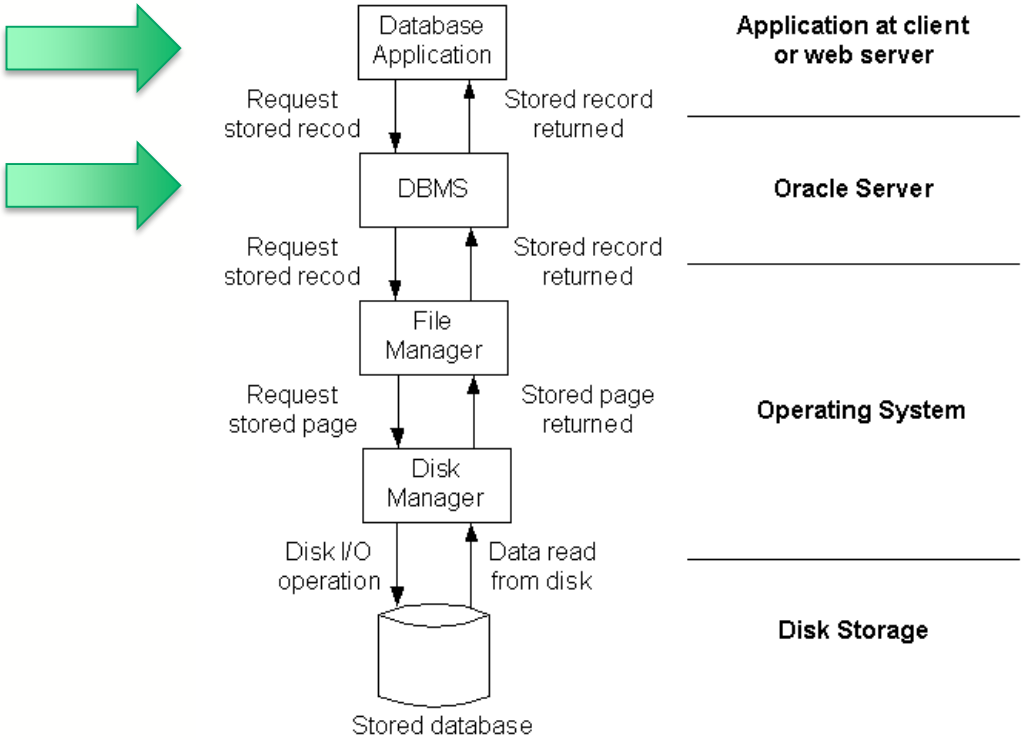
Database Design Process

- Physical Database Design



Application, DBMS, and I/O Process

- Data Retrieval Process Via a Database Application



Record Storage

- **Computer storage medium:**
  - Primary storage: main memory, cache memory for fast data access.
  - Second storage: magnetic disks, optical disks, tapes, and drums. They have a much larger capacity, are less expensive, but slower access than that of primary storage.
  - Flash memory (sometimes called "flash RAM") is a type of constantly-powered nonvolatile memory that can be erased and reprogrammed in units of memory called *blocks.*

Disk Devices

- The physical divisions of a disk are cylinders (for a disk pack), tracks, sectors (arc of a track), and blocks (see Figures 16.1 and 16.2).
- A disk is a random access addressable device. Data transfer between main memory and disk is in units of *blocks*.
- The process has seek time (locating the correct track by the head), rotational delay (pointing to the right block), and block transfer time in milliseconds. Locating data on a disk and transferring a disk block is in the order of milliseconds while CPU processing time is in the order of microseconds to nanoseconds.

Computer Hardware Improvement Rates

- **Computing component improving rate comparison**

Computing components	Improving rate/year	Comments
RAM capacities	133% to 200%	10^{-9} sec.
CPU	33% to 50%	
Disk access time	Less than 10%	Major slowdown
Disk transfer rate	20%	Major slowdown
Disk capacities	50%	

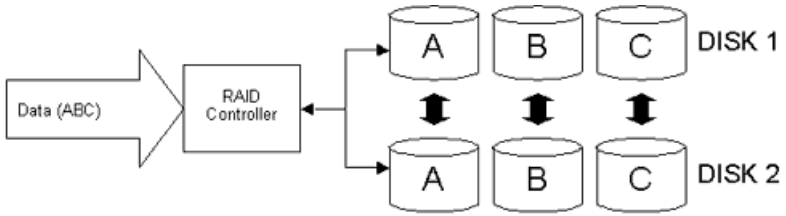
Disk I/O remains the bottleneck for HW improvement
(Registers (1-9 x 10^{-9} sec.), cache (10-40 x 10^{-9} sec.), Disk (2-3 x 10^{-3} sec.))

Redundant Array of Independent Disks (RAID)

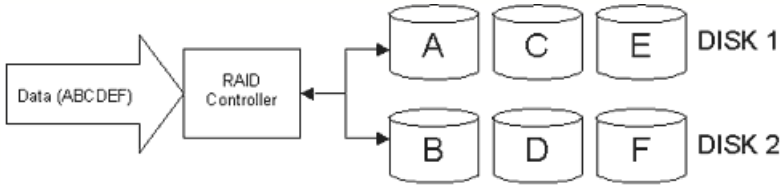
- RAID is used for performance improvement and redundancy.
- Major objectives are to improve disk performance and reliability.
  - Improves performance (employs parallelism concepts to improve access time)
  - Improves reliability (uses redundancy and error checking codes)
  - Increases storage capacity since it is viewed as a large logical disk (comprised of several physical hard drives)

Redundant Array of Independent Disks (RAID) (cont.)

- **RAID Concepts**
  - Logical drive encapsulating several physical drivers
  - Mirroring or Shadowing



  - Data Striping



  - Parallelism, Load Balancing, and Parity

Redundant Array of Independent Disks (RAID) (cont.)

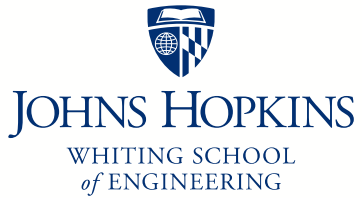
- RAID does NOT protect against power failures, operator mistakes, buggy RAID software, hardware failure, virus attacks on the storage system, inadvertently deleting files, or lightning hitting the building and causing a fire in the computer room.
- RAID is not a substitute for regularly scheduled maintenance and backup procedures.

Johns Hopkins Engineering

Principles of Database Systems

Module 11 / Lecture 2

File Structures and Indexes



Data Types, Files, and Records

- Data Types in a Record:
  - Data is stored in the form of records. Each record consists of a collection of related data values or items.
  - Each value corresponds to one of the basic data types: (Boolean, integer, string of characters, date) and advanced data types: (user-defined, pointers to external files, VARRAY, references, large objects (LOB), XML, and JSON.

- Files
  - A collection of records stored in a physical storage with the same format.

- Fixed/variable-length records.

Unspanned Versus Spanned Record Blocks

- An unspanned record block may contain multiple records and leave unused space at the end of the block.
- A spanned record block contains a pointer linking to the next block to store records.

Allocating File Blocks on Disk

- Continuous allocation: easy to access data blocks, hard to expand to new records.
- Linked allocation: easy to expand but is slow to access all data blocks.
- A combination of both allocates clusters (also called segments or extents) of consecutive blocks, and clusters are linked together.
- Indexed allocation: index blocks with indexes pointing to the actual data block.

Headers on File and Block

- File information is stored in a file header including the disk addresses of the file blocks, record format descriptions, and so on. Overhead causes less data storage space available in a data block.

- The purpose of a good file organization is to locate the needed block(s) by the DBMS with a minimal number of block transfers in order to improve the performance.

Operations on Files

- Basic operations include ‘insert’, ‘retrieve’, ‘update’, and ‘delete’ records or files.
- File organization concerns the organization of data file into records, blocks, and access structures. How are they stored in the physical storage on the disk and how are they interlinked?
- The access method uses different programs to perform basic database operations.
- Performance is the primary concern for file organization and access method.

Physical Database Design

- Database is implemented on the secondary storage
  - Database schema – database objects, constraints, triggers
  - File organizations – format and types
  - Indexes – performance
  - Size estimates
  - Security requirements – data and access control via system, application, or DBMS protection

Files of Unordered Records (Heap Files)

- A query for a record(s) uses a linear/sequential search if there is no access path on the file.
- A file of unordered fixed-length records using unspanned blocks and contiguous allocation, *i-th* record's location can be easily identified if the *i* value is known. However, the value does not relate to the search condition. Therefore, this type of file requires an access path using indexes.

Files of Ordered Records (Sorted Files)

- A query for a record is extremely efficient using a sorted key. Searching the next record is usually stored at the same block or the next block.
- A binary search based on the blocks rather than on the records.
- In general, a binary search reads for $log_2 n$ blocks comparing $n/2$ for an unordered file.

Hashing File Organization

- Using the hashing function with the value of hash field to generate the address of the disk block (hash address) in which the record is stored. Only the equality search condition can be applied. The search is based on hash field of the file and is very fast.

- For example, employee_id ranges 1-1000, the hashing address = f(employee_id) where f() = remainder (employee_id / 1000)

Hashing File Organization (cont.)

- There is a possibility that different values of a hash field have the same hashing address.
- The number of possible values of a hash field (hash field space) usually is much larger than that of available addresses for records (address space) to save unused spaces.
- A collision occurs when inserting a second record with a hashing address is used by another record.

Hashing File Organization (cont.)

- A good hashing function should distribute the records uniformly over the assigned spaces and minimize collisions.
- The process requires choosing a hashing function with a method for collision resolution.
- A suitable method for external hashing is the bucket technique (see Figure 16.9) and with chaining for bucket overflow caused by collisions (see Figure 16.10).

Hashing File Organization (cont.)

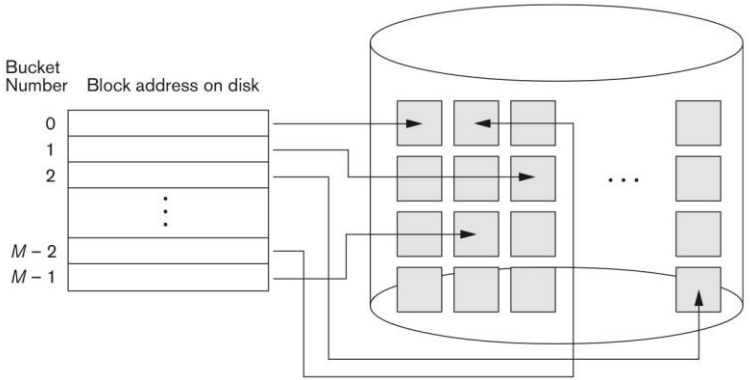


Figure 16.9 Matching bucket numbers to disk block addresses.

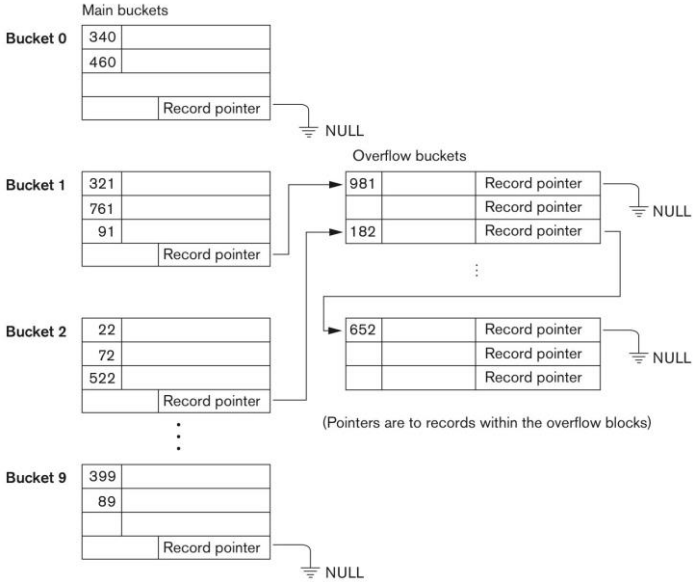


Figure 16.10 Handling overflow for buckets by chaining.

Johns Hopkins Engineering

Principles of Database Systems

Module 11 / Lecture 3

File Structures and Indexes

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Index Structure for Files

- **Types of Single-level Ordered Indexes:**
  - The major advantage of an index is that it speeds up retrieval.
  - The update process is slowed down. When a new record is stored into a table, the index(es) for that record must also be added to the corresponding index file(s).

Primary Indexes

- **Primary Indexes:**
  - A primary Index is an index specified in the ordering key field of an ordered file of records (see Figure 17.1).
  - The ordering key field is called the primary key of a data file, and records can be uniquely identified.
  - An index file in which each entry (each record) consists of two values, the value of the primary key field for the first record in the block, and a pointer to that block.

Primary Indexes (cont.)

- **Primary Indexes (cont.):**

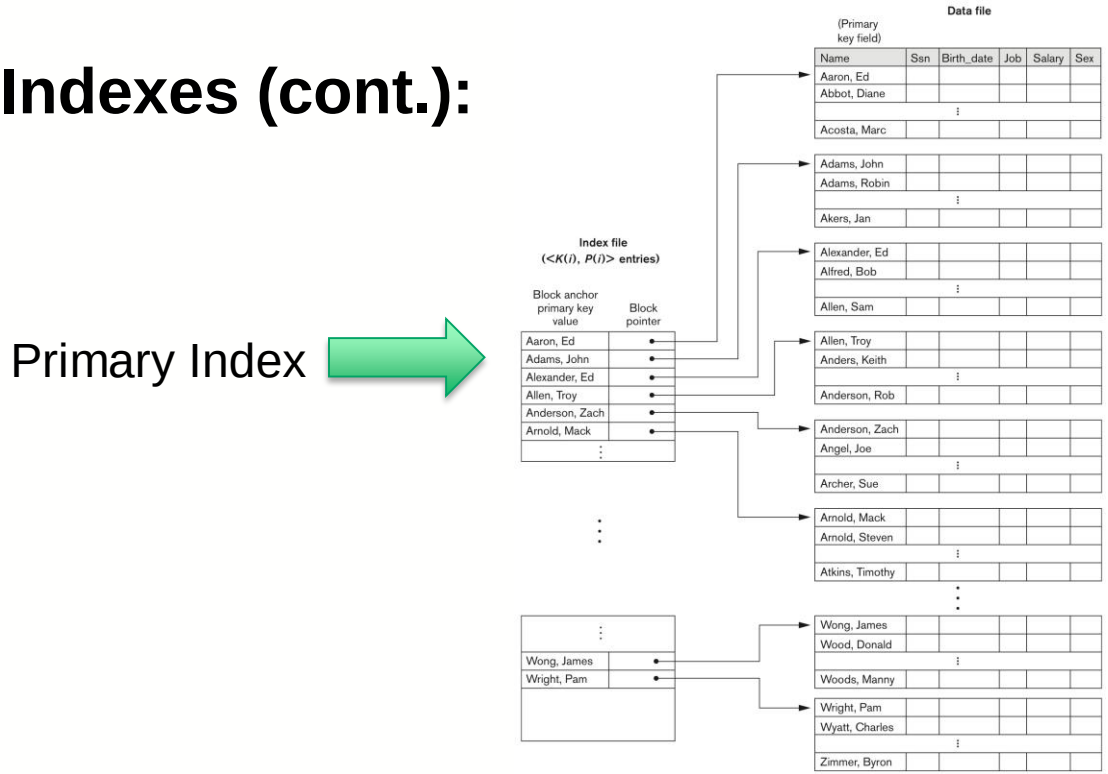


Figure 17.1 Primary index on the ordering key field of the file shown in Figure 16.7.

Primary Indexes (cont.)

- **Primary Indexes (cont.):**
  - The first record in each block of the data file is called the anchor record of the block.
  - A nondense index contains only one entry for each disk block of the data file. A dense index contains an entry for every record in the file.
  - A primary index is a nondense index.
  - Binary search can be applied on the index file (see Exercise 1 with 30K records – linear, binary and primary index searchers.)

Clustering Indexes

- **Clustering Indexes:**
  - The file records are physically ordered in a nonkey field without distinct value for each record.
  - The column of a table in a cluster is called the cluster field and a clustering index is created.
  - Clustering index is a non-dense index. It has an entry for every distinct value of the indexing field (see Figure 17.2).
  - Chaining data blocks resolves the inserting record problem (see Figure 17.3).

Clustering Indexes (cont.)

- **Clustering Indexes (cont.):**

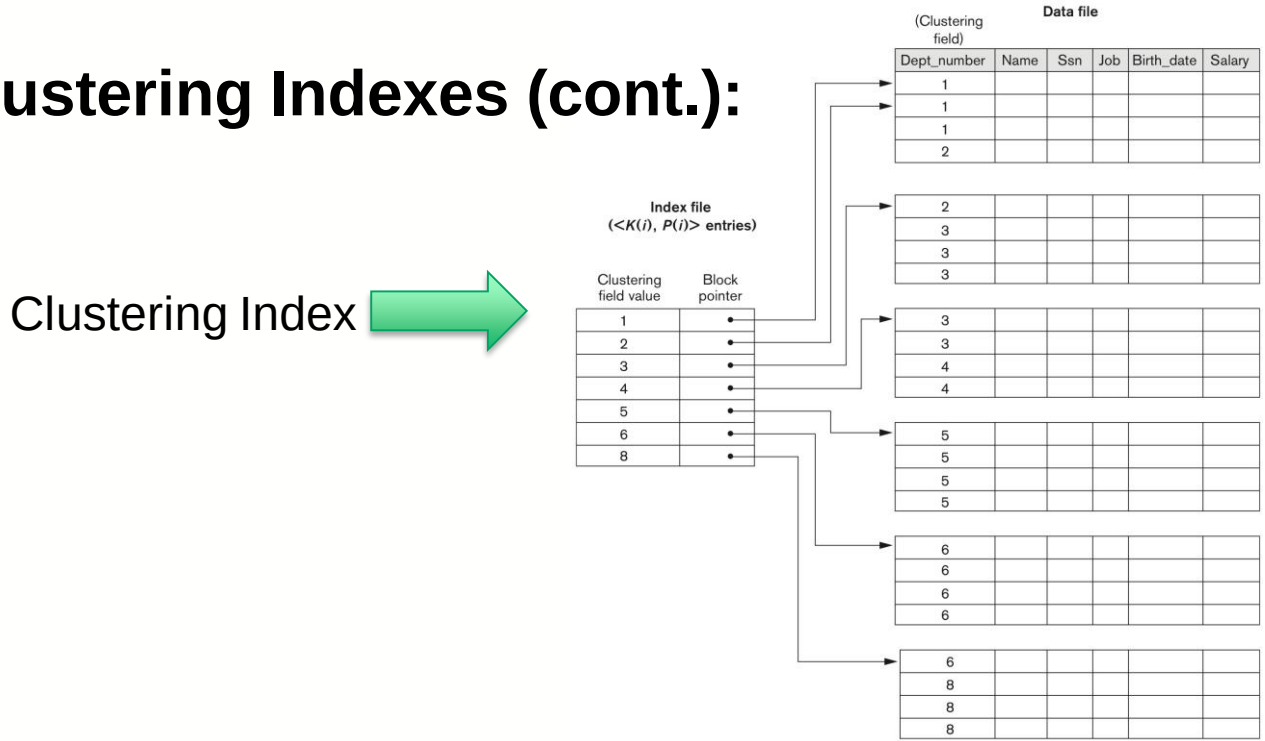


Figure 17.2 A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Secondary Indexes

- **Secondary Indexes:**
  - A secondary index file provides alternative ways to access a file. It contains two fields. The first field is of the same data type as a non-ordering field of the data file, and is called an indexing field of the file. The second field can be a block pointer or a record pointer.
  - Many secondary indexes can be used to a data file.

Secondary Indexes (cont.)

- **Secondary Indexes (cont.):**

A Dense Secondary Index

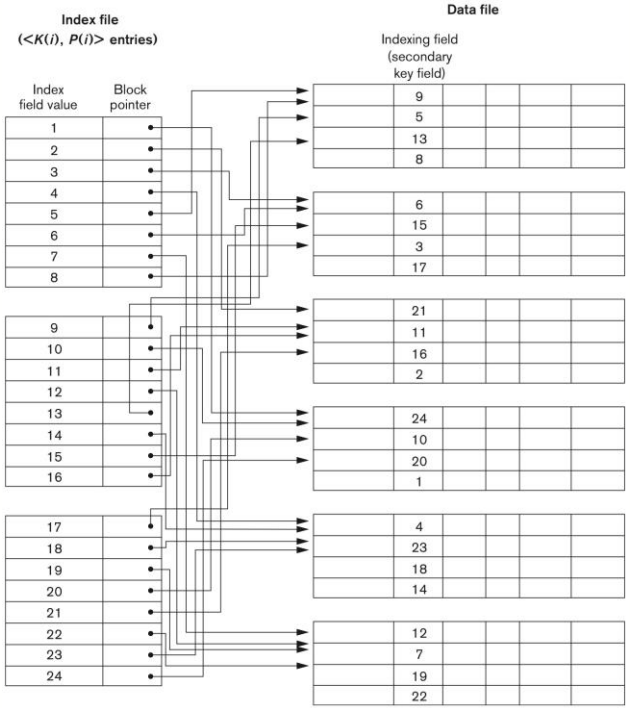


Figure 17.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

Multilevel Indexes

- **Multilevel Indexes:**
  - Multilevel indexes are used to reduce the number of blocks accessed when searching indexes. The number of blocks needed to search a record is substantially reduced by a factor of ***1/fo***, where ***fo*** is called *fan-out* of the multilevel index.
  - A multilevel index uses the key value of an ordered file as the first level of a multilevel index. The index for the first level is called the second level of index (see Figure 17.6).

Multilevel Indexes (cont.)

- **Multilevel Indexes (cont.)**

A Two-level Primary Index

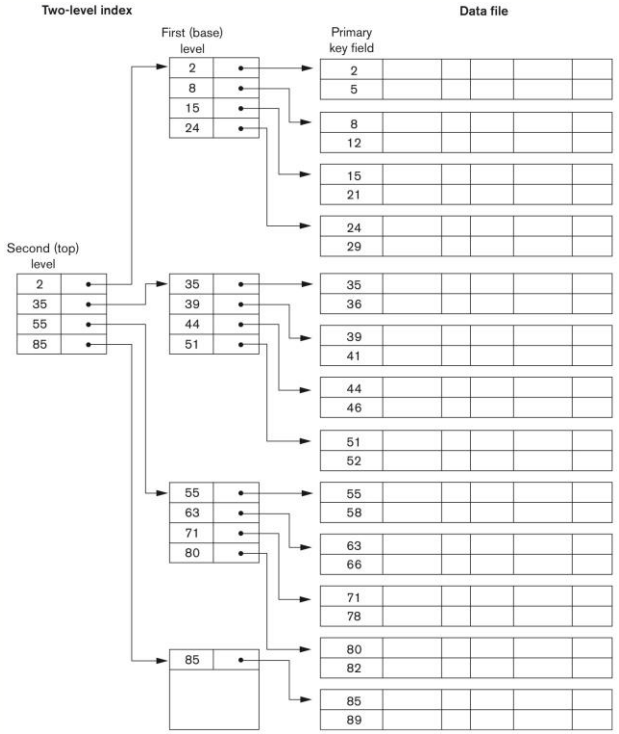


Figure 17.6 A two-level primary index resembling ISAM (indexed sequential access method) organization.

Johns Hopkins Engineering

Principles of Database Systems

Module 11 / Lecture 4

File Structures and Indexes

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Dynamic Multilevel Indexes Using B-Trees

- **Dynamic Multilevel Indexes Using B-Trees:**
  - A tree consists of nodes. Each node has zero or more child nodes and one parent node, except for the root node (see Figure 17.7).

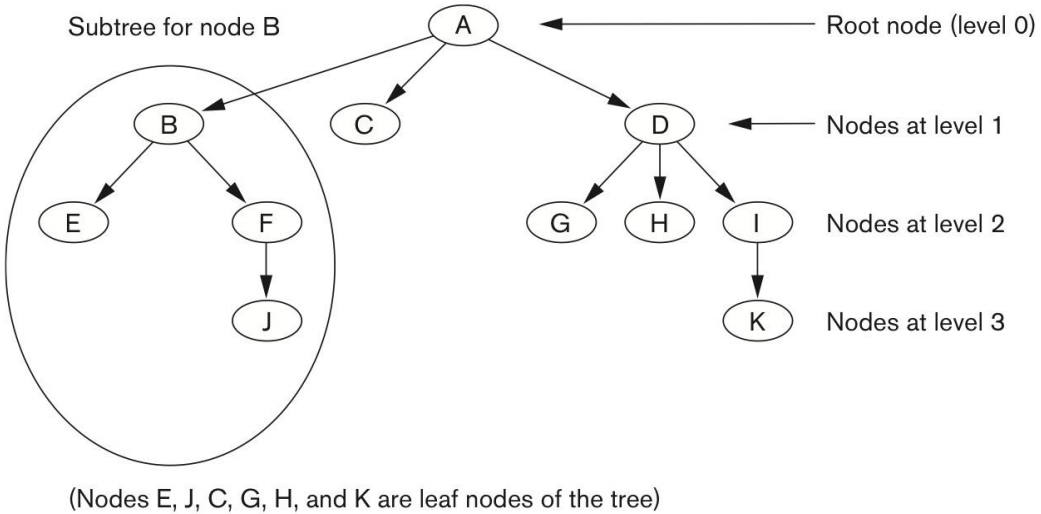


Figure 17.7 A tree data structure that shows an unbalanced tree.

Dynamic Multilevel Indexes Using B-Trees (cont.)

- **Dynamic Multilevel Indexes Using B-Trees (cont.):**
  - 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 $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, where q ≤ p. Each P_i is a pointer to a child node; each K_i is a search value. Two node constraints:
    - $K_1 < K_2 < \ldots < K_{q-1}$
    - All values X in the subtree pointed at by P_i, $K_{i-1} < X < K_i$
  - B-tree structure is similar to the search tree with tree pointers P_i, search key fields K_i, and data pointers Pr_i.
  - B-tree has additional constraints to ensure that it is balanced.

Dynamic Multilevel Indexes Using B-Trees (cont.)

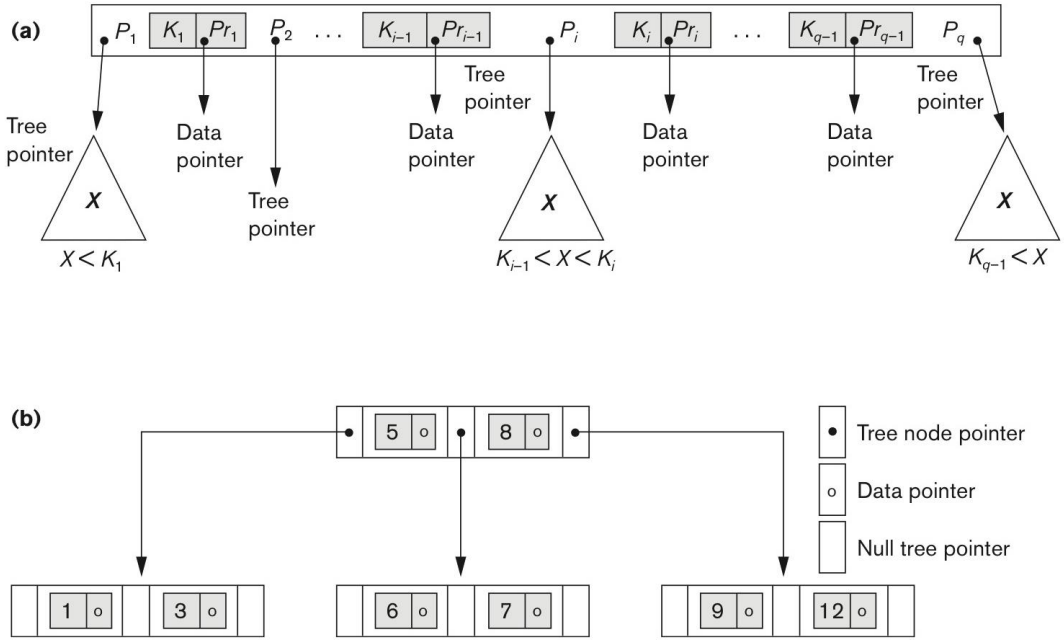


Figure 17.10 B-tree structures. (a) A node in a B-tree with q − 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Dynamic Multilevel Indexes Using B^+-Trees

- **Dynamic Multilevel Indexes Using B^+-Trees:**
  - Data pointers are stored only at the leaf nodes of the tree.
  - The leaf nodes have an entry for every value of the search field, along with a data pointer to the record if the search field is a key field.
  - The leaf nodes are linked together for ordered access on the search field to the records.

Dynamic Multilevel Indexes Using B+-Trees (cont.)

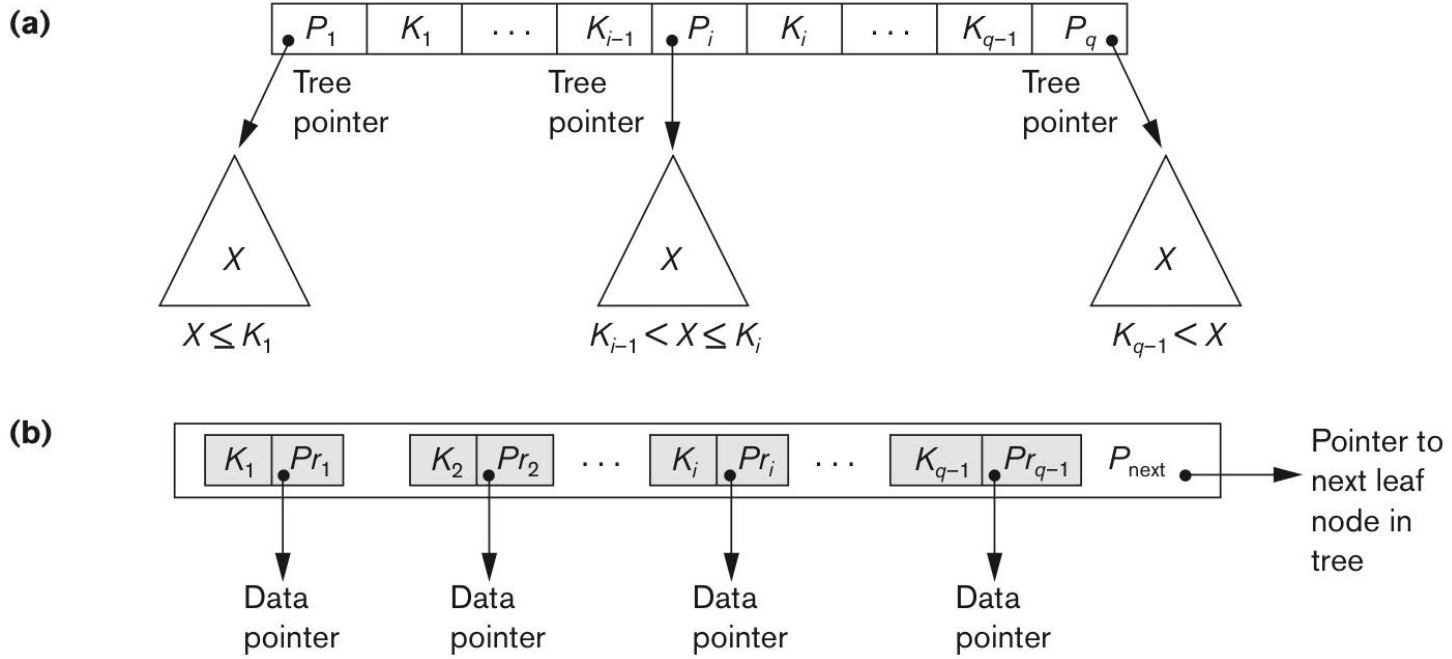


Figure 17.11 The nodes of a B+-tree. (a) Internal node of a B+-tree with q − 1 search values. (b) Leaf node of a B+-tree with q − 1 search values and q − 1 data pointers.

Dynamic Multilevel Indexes Using B+-Trees (cont.)

- **Dynamic Multilevel Indexes Using B^+-Trees (cont.):**
  - The internal nodes include search values and tree pointers without any data pointer to build other level(s) of multilevel indexes. More entries can be packed into an internal node of a B^+-tree than for a similar B-tree (see Examples 6 and 7).
  - Because there is no single storage structure that is optimal for all applications, B-tree and B^+-tree provide the best all-around performance.

Benefits and Trade-offs of Creating Indexes

- Overhead involved in maintenance and use of secondary indexes that has to be balanced against performance improvement gained when retrieving data.

- Create an index on a column, if:
  - The column is used frequently in WHERE clauses or in a join condition.
  - Every value within the column is unique.
  - The column contains a wide range of values (high cardinality column).

Benefits and Trade-offs of Creating Indexes (cont.)

- A function-based index is an index built on an expression.
- A function-based index precomputes the value of a computationally intensive function and stores it in the index. An index can store computationally intensive expression that you may access often

```
CREATE INDEX Idx ON Example_tab(Col_a + Col_b);
SELECT * FROM Example_tab WHERE Col_a + Col_b < 10;
```

- The cost-based optimizer may prefer to scan a table instead of using the indexes.

Benefits and Trade-offs of Creating Indexes (cont.)

- A composite key (multicolumn index) is a primary key that consists of more than one column.
  - The order of the columns in a composite index is important and it also depends upon how your queries are written.
  - A composite index has the left-most (leading) column that occurs most often in the queries.
  - If the leading column is more selective, it can reduce the query cost.

```
SQL> count(*), count(distinct col_1), count(distinct col_2), count(distinct col_3)
  2  FROM test_table;
```

Benefits and Trade-offs of Creating Indexes (cont.)

- **Do not create indexes, if:**
  - The table is small.
  - Most queries are expected to retrieve more that 10-15% of the rows.
  - The table is updated frequently.
  - A similar index is created (e.g., Index1 on (C1) and Index2 on (C1, C2)).

  Good index design balances query performance and system maintenance cost.

Types of database applications

- **Online Transaction Processing (OLTP)**
  - May involve many simple DMLs
  - Determine required indexes
  - Minimize the number of OLTP indexes
  - May have complex queries for statistical reports
  - Use EXPLAIN PLAN to show how a statement will be processed (by the optimizer)

Types of database applications (cont.)

- **Data Warehouse**
  - Consolidate view of enterprise data from various databases
  - May design and use to support decision making through Online Analytical Processing (OLAP analysis) and data mining
  - Is optimized for reporting and analysis
  - May involve complex queries for processing a large amount of data
  - Use tools to analyze user queries

Johns Hopkins Engineering

Principles of Database Systems

Module 11 / Lecture 5

File Structures and Indexes

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Oracle Index Implementations

- **Using B^+-tree (B*-tree)**
  - An index file in which each entry (each record) consists of two values, a key value (a data value) and a pointer (ROWID).
  - Every row in a table of an ORACLE database is assigned a unique ROWID (a pseudo-column) that corresponds to the physical address of a row's row piece (the initial row piece if the row is chained among multiple row pieces).

Oracle Index Implementations (cont.)

- **ROWID Format with a *10-byte* address:**

  OOOOOO - Data Object #

  FFF - Relative File #

  BBBBBB - Block #

  SSS – Slot #

  A base-64 encoding with the characters A through Z, a through z, 0 through 9, + and /.

  Example: Show ROWID and ename in emp table

```
SQL> SELECT ROWID, ename FROM emp;
       ROWID              ENAME
       ------------------ ----------
       AAAAVJAAEAAAABEAAA SMITH
       AAAAVJAAEAAAABEAAB ALLEN
       AAAAVJAAEAAAABEAAC WARD
       . . .
       ------------------ ----------
       D Obj#RF#Blk  #SL#
```

Index-organized Table

- **Index-organized Table:**
  - An *index-organized table* (IOT) with the data for a table is held in its associated index.
  - IOT provides faster key-based access to table data for queries.
  - The data rows are built on the primary key (must be specified) for the table, and each B*-tree index entry contains:

    `<primary_key_value, non_primary_key_column_values>`

Bitmap Index

- **Bitmap Index:**
  - Indexes designed for use in large, slow-moving systems (e.g., data warehouse features for decision support) including columns with low distinct values.

    Query Example:

    Tell me about the customers living in the DC suburbs, who own 2 cars, have 3 children, live in a 4-bedroom house.

    - Any single condition in this search may return a very large number of rows, no need to create B*-tree indexes on any specific combination of columns.
    - Full table scan.

Bitmap Index (cont.)

- **Bitmap Index (cont.):**
  - How bitmap index work:
    - In a *bitmap index*, a bitmap for each key value is used instead of a list of rowids.
    - A bitmaps for Cars (0, 1, 2, ...) and Children (0, 1, 2, 3, …), and Bedrooms (0, 1, 2, 3, 4, …) respectively.
    - Using the Bitwise “AND” operator to filter qualified rows (key values).
    - Bitmap indexes are particularly useful for queries that contain [NOT] EXIST, UNIQUE, or GROUP BY on low cardinality columns.

Bitmap Index (cont.)

- **Bitmap Index (cont.):**
  - Benefits for using bitmap index:
    - Reduced response time for large classes of ad hoc queries.
    - A substantial reduction of space usage compared to other indexing techniques.
    - Dramatic performance gains even on very low end hardware.
  - Bitmap indexes are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Table and Index Partitions

- Very large tables and indexes can be partitioned into smaller, more manageable pieces.
  - Improve performance with focused data access, and speed up data access from minutes to seconds.
  - Increase availability to critical information.
  - Improve manageability over data.

Table and Index Partitions (cont.)

- Proper information lifecycle management
  - As data ages, activity of the data declines and overall volume grows (e.g., business sales, bank transactions)
    - Current month data – most active data set (small size) stored in high performance storage
    - Previous month data – less active data set (medium size) stored in low cost storage
    - Previous year data – historical data set (large size) stored in cheap cost storage
    - Previous years data – archive data set (huge size) stored in offline storage
- Partitioning by range via a partition key (e.g., number or date), by list of values, by hash

Table and Index Partitions (cont.)

- Multiple columns are allowed.

```
… PARTITION BY RANGE (year, month, day)
(PARTITION p1 VALUES LESS THAN (2016, 12, 31) TABLESPACE ts201612),
 PARTITION p2 VALUES LESS THAN (2017, 12, 31) TABLESPACE ts201712),
 ......,
 PARTITION pOutRange VALUES LESS THAN (MAXVALUE, MAXVALUE, MAXVALUE)
    TABLESPACE tsOutRange);
```

- A partitioned table can have partitioned or non-partitioned indexes.
- The major reason for using range partitioning is the tremendous performance benefit (e.g., query, parallel DML, import/export).

Table and Index Partitions (cont.)

- Backup and recovery process increases overall availability.
- DBMS handles the access methods to reduce amount of touched data (IO reduction) to speed up response time or improve throughput.
- Partitions are transparent to developers and database applications.
- Partitions can have local or global indexes.
- Partitions are useful for very large databases and high performance and availability requirements.

Performance tuning for VLDB

- A CPU bound database server (examining top 5 timed events)
  - Not easy to tune computationally intensive applications
  - Using faster CPU and parallel processing
- An I/O bound database server
  - Add more RAM buffers for database usage and SQL tuning are the best way to reduce disk I/O
- Using 64-bit technology
  - Improving RAM addressing; Faster CPU and file I/O; High parallelism with multiple CPUs
- NoSQL for big data with performance, scalability, flexible schema advantages