

Information Systems

Chapter 4:

Storage and File Structures

Rita Schindler | TU Ilmenau, Germany

www.tu-ilmenau.de/dbis

Storage and File Structures

- ▶ Overview of Physical Storage Media
- ▶ Buffer Management
- ▶ File Organization
- ▶ Indexing

Motivation (I)

- ▶ Database systems always involve secondary storage – the disk and other devices that store large amounts of data that persists over time.
- ▶ This chapter summarizes what we need to know about how a typical computer system manages storage.
 - ▶ We review the memory hierarchy of devices with progressively slower access but larger capacity.
 - ▶ We examine disks in particular and see how the speed of data access is affected by how we organize our data on the disk.
 - ▶ (We also study mechanisms for making disks more reliable.)

Motivation (2)

- ▶ Then, we turn to how data is represented.
- ▶ We discuss the way tuples of a relation or similar records or objects are stored.
 - ▶ Efficiency, as always, is the key issue.
- ▶ We cover ways to find records quickly, and how to manage insertions and deletions of records, as well as records whose sizes grow and shrink

Classification of Physical Storage Media

- ▶ **Main criteria:**
 - ▶ Speed with which data can be accessed and
 - ▶ price (cost) per unit of data of a storage medium, i.e. cost per unit of data to buy medium
- ▶ **Reliability**
 - ▶ data loss on power failure or system crash
 - ▶ physical failure of the storage device
- ▶ **Classification:**
 - ▶ **volatile storage:** loses contents when power is turned off
 - ▶ **non-volatile storage:** contents persist when power is switched off. Includes secondary and tertiary storage, as well as battery-backed main memory.

Physical Storage Media

▶ **Cache:**

- ▶ fastest and most costly form of storage;
- ▶ volatile;
- ▶ managed by the hardware and/or operating system

▶ **Main Memory:**

- ▶ general purpose instructions operate on data in main memory
- ▶ fast access but in general too small to store the entire database (or even an entire relation)
- ▶ volatile – content of main memory is usually lost if a power failure or system crash occurs

Physical Storage Media (2)

▶ **Magnetic-Disk Storage:**

- ▶ primary medium for the long term storage of data
- ▶ typically stores the entire database (i.e., all relations and associated access structures)
 - ▶ data must be moved from disk to main memory for access and written back for storage (**insert, update, delete, select**)
 - ▶ direct-access, i.e., it is possible to read data on disk in any order
 - ▶ usually survives power failures and system crashes; disk failure, however, can destroy data, but is much less frequent than system crashes

▶ **Optical Storage:**

- ▶ non-volatile;
- ▶ CD-ROM/DVD most popular form;
- ▶ Write-Once-Read-Many (WORM) optical disks are typically used for archival storage.

Physical Storage Media (3)

▶ **Tape Storage:**

- ▶ non-volatile;
- ▶ used primarily for backup and export (to recover from disk failures and to restore previous data), and for archival data
 - ▶ sequential access, much slower than disk
 - ▶ very high capacity (GB tapes are common)
 - ▶ tape can be removed from drive → storage cost much cheaper than disk.

Hierarchy

- ▶ The devices with smallest capacity also offer the fastest access speed and have the highest cost per byte
- ▶ Levels from the lowest (fastest-smallest) up:
 - ▶ Cache
 - ▶ Data and instructions are moved to cache from main memory when they are needed by the processor
 - ▶ Can be accessed in a few nanoseconds
 - ▶ Main memory
 - ▶ Everything that happens in the computer – instruction executions, data manipulations – as working on information that is resident in main memory
 - ▶ Secondary Storage
 - ▶ Tertiary Storage

Storage Hierarchy

Storage	Level	Access speed	volatail	also called ...	Examples
Primary	high	fastest	volatile		cache, main memory
Secondary	next lower	moderately fast access time	non-volatile	on-line storage	magnetic disks
Tertiary	lowest	slow access time	non-volatile	off-line storage	optical disks, tapes

Transfer of Data Between Levels

- ▶ Accessing desired data or finding the desired place to store data takes a *great deal of time*
 - ▶ → each level is organized to transfer large amounts of data to or from the level below
- ▶ The disk is organized into disk **blocks** (*pages* in operating systems)
- ▶ Entire blocks are moved from or to the main memory (buffer)

⇒ transfer unit: blocks

Mechanics of Disks

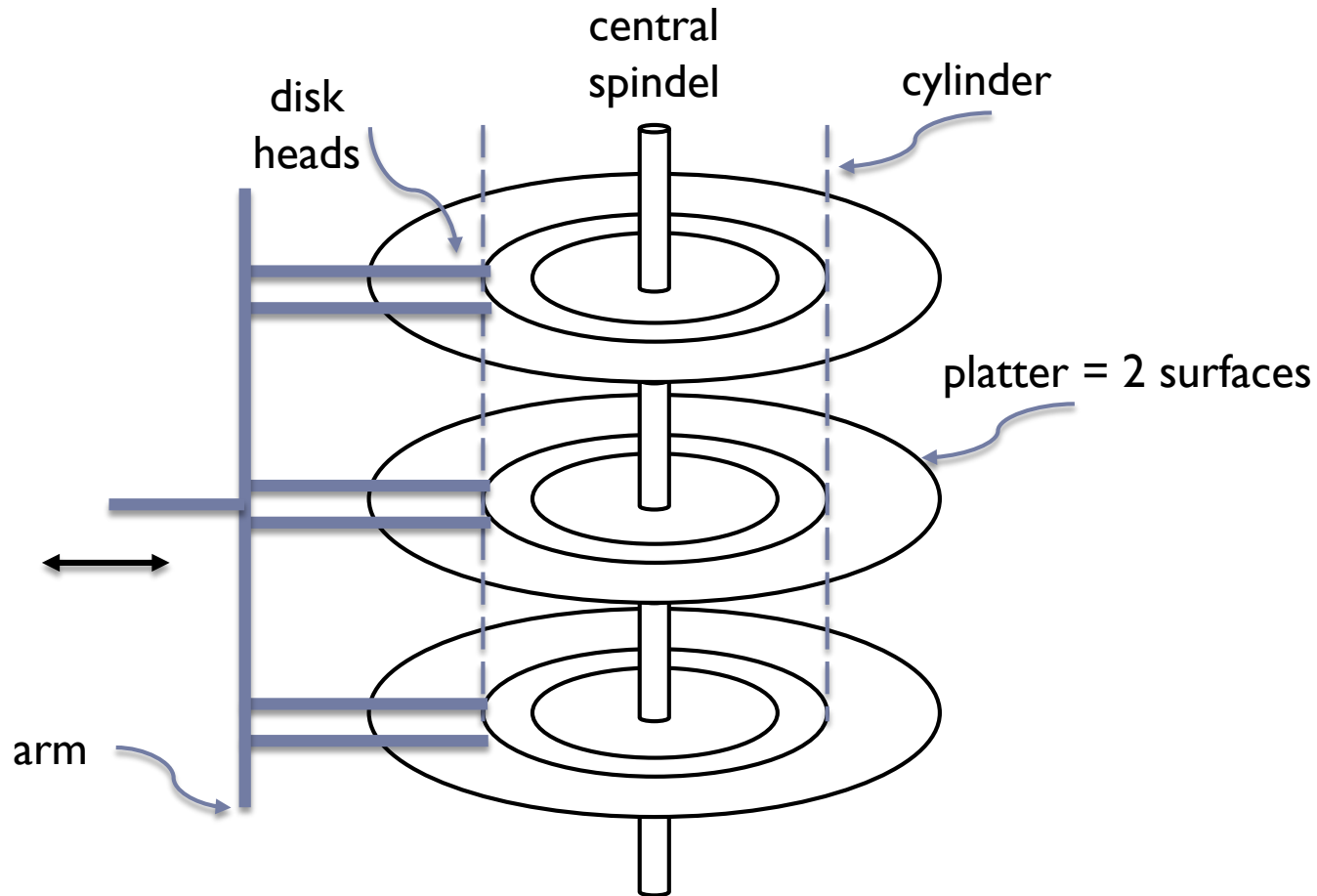


Fig. 1: A typical disk

Mechanics of Disks

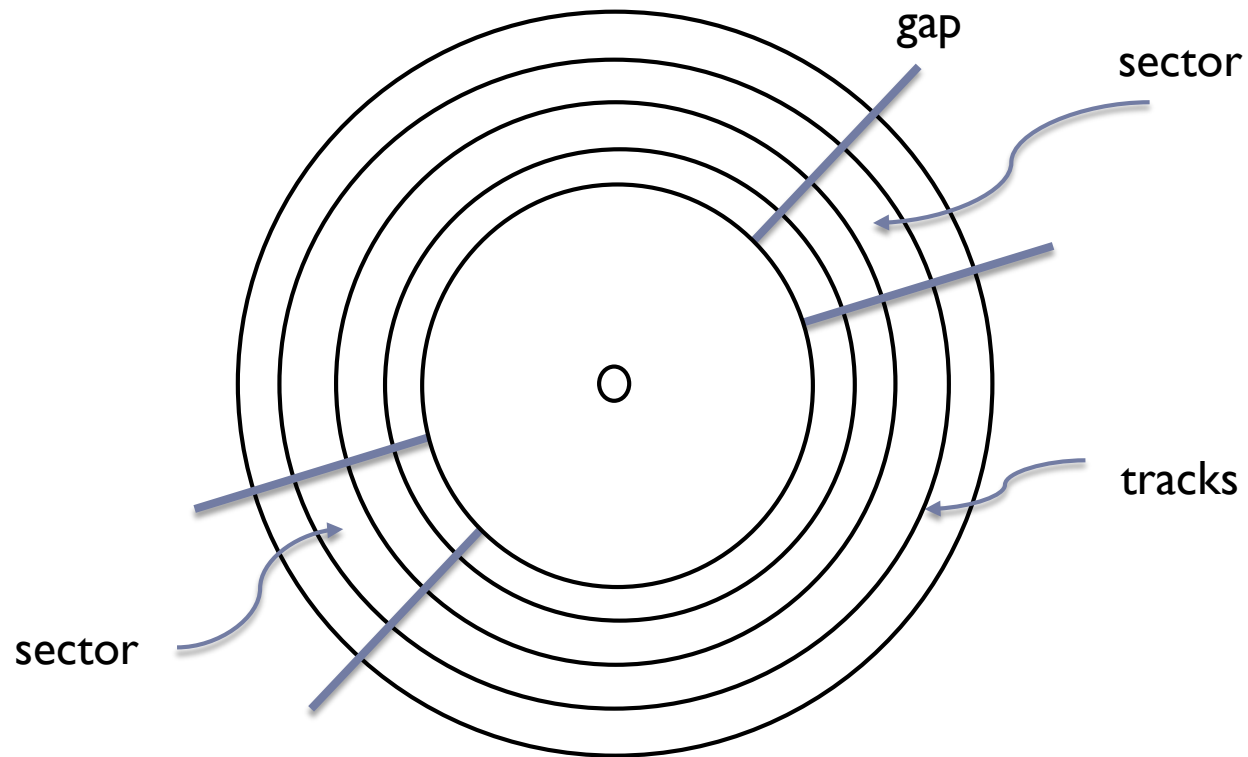


Fig. 2: Top view of a disk surface

Terms

Two principal moving pieces of a disk drive:

- ▶ **disk assembly**: consists of one or more circular **platters** that rotate around the central spindle
 - ▶ organized into **tracks**: concentric circles on a single platter
 - ▶ tracks that are at a fixed radius from the center, among all the surfaces, form one **cylinder**
 - ▶ tracks are organized into **sectors**, which are segments of the circle separated by **gaps** that are not magnetized
- ▶ **head assembly**: holds and moves the **disk heads**
 - ▶ heads are each attached to an **arm**,
 - ▶ and the arms for all the surfaces move **in** and **out** together

Disk Access Characteristics

Accessing a block requires three steps (each has an associated delay):

1. Disk controller positions the head assembly at the cylinder containing the track on which the block is located → **seek time**
2. ... waits while the first sector of the block moves under the head → **rotational latency**
3. All sectors and the gaps between them pass under the head, while the disk controller reads or writes data in the sectors → **transfer time**

Latency of the disk = seek time + rotational latency + transfer time

- ▶ **Mean time to failure (MTTF):** the average time the disk is expected to run continuously without any failure.

Optimization of Disk-Block Access

- ▶ **Block** – a contiguous sequence of sectors from a single track
 - ▶ Data is transferred between main memory and disk in the form of blocks
 - ▶ Block size ranges from 512 bytes to several kilobytes
 - ▶ Non-volatile buffers speed up disk writes by immediately writing blocks to a non-volatile RAM buffer; controller then writes to disk whenever the disk has no other requests.
- ▶ File organization – **optimize** block access time by organizing the blocks to correspond to how data will be accessed (e.g., store related information **on the same** or nearby **cylinder**).
 - ▶ Seek time represents about half the time it takes to access a block

⇒ **storage unit: blocks**

Storage Access

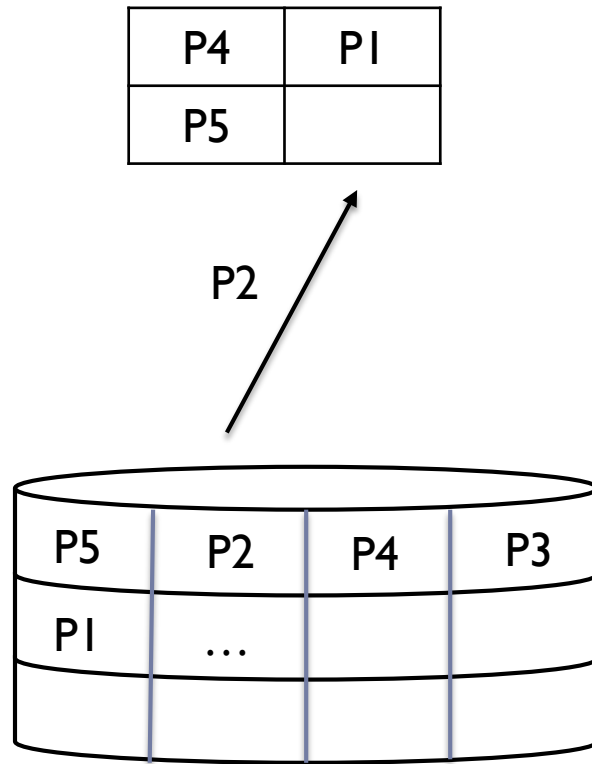
- ▶ A database file is partitioned into fixed-length storage units called **blocks**.
 - ▶ Blocks are units of both storage allocation and data transfer.
- ▶ Database system seeks to minimize the number of block transfers between disk and main memory. Transfer can be reduced by keeping as many blocks as possible in the main memory.
- ▶ **Buffer**: Portion of the main memory available to store copies of disk blocks.
- ▶ **Buffer Manager**: System component responsible for allocating and managing buffer space in main memory.
 - ▶ several replacement policies

Buffer Manager

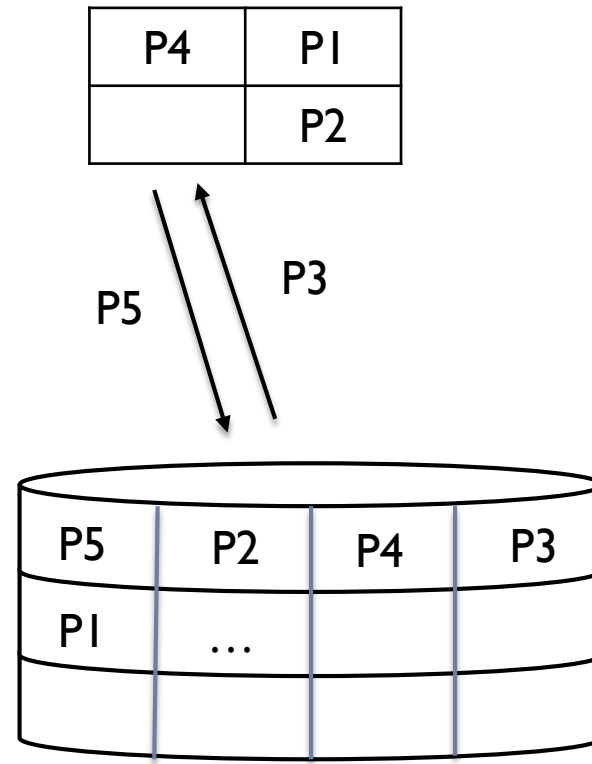
Program calls on buffer manager when it needs a block from disk

- ▶ The requesting program is given the address of the block in main memory, if it is already present in the buffer.
- ▶ If the block is not in the buffer, the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other blocks, if necessary to make space for new blocks.
- ▶ The block that is thrown out is written back to the disk only if it was modified since the most recent time that it was written to/fetched from the disk.
- ▶ Once space is allocated in the buffer, the buffer manager reads in the block from the disk to the buffer, and passes the address of the block in the main memory to the requesting program.

Buffer Management



a) Read P2



b) Replace P5

Arranging Data on Disk

- ▶ A data element (tuple, object) is represented by a **record**, which consists of consecutive bytes in some disk block
 - ▶ Collections (relations) are represented by placing the records of the elements **in one or more blocks (→ files)**
 - ▶ It is normal for a block to hold only elements of one relation
 - ▶ Although there are organizations where blocks hold tuples of several relations
- ⇒ Basic layout techniques for both records and blocks

File Organization

- ▶ A database is stored as a collection of **files**.
- ▶ Each file is a sequence of **records**,
- ▶ and a record is a sequence of **fields**.

- ▶ Approaches to organizing records in files:
 - ▶ Assume the record size is fixed.
 - ▶ Each file has records of one particular type only.
 - ▶ Different files are (typically) used for different relations.

Fixed-Length Records

- ▶ Simple approach:

- ▶ Store record i starting at byte $n * (i - 1)$, where n is the size of each record.
- ▶ Record access is simple but records may span blocks.

- ▶ Deletion of record i (alternatives to avoid fragmentation)

1. move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 2. move record n to i
- ▶ Link all free records in a *free list*

Free Lists

- ▶ Maintained in the block(s) that contains deleted records.
- ▶ Store the address of the first record whose content is deleted in the file header.
- ▶ Use this first record to store the address of the second available record, and so on.

One can think of these stored addresses as **pointers**, because they “point” to the location of a record. (linked list)

- ▶ More space efficient representation: reuse space for normal attributes of free records to store pointers (i.e., no pointers are stored in in-use records).
- ▶ Requires careful programming: **Dangling pointers** occur if a record is moved or deleted and another record contains a pointer to this record; that pointer then doesn't point any longer to the desired record.

Variable-Length Records

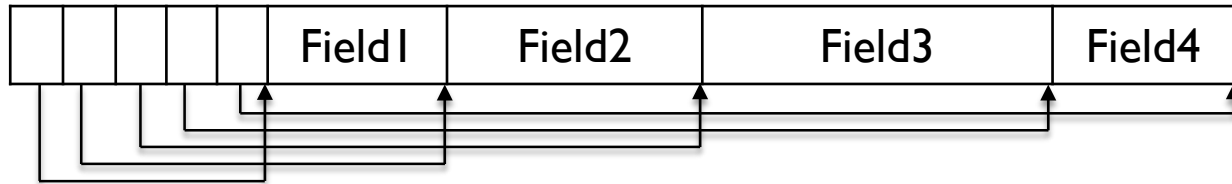
- ▶ Variable-length records arise in database systems in several ways:
 - ▶ Storage of **multiple record types** in a file.
 - ▶ Record types that allow **variable length** for one or more **fields** (e.g., **varchar**)
- ▶ Approaches to store variable length records:
 - ▶ End-of-Record marker
 - ▶ difficult to reuse space of deleted records (fragmentation)
 - ▶ no space for a record to grow (e.g., due to an update) → requires moving
 - ▶ Field delimiters (e.g., a \$)

Field1	\$	Field2	\$	Field3	\$	Field4	\$
--------	----	--------	----	--------	----	--------	----

- ▶ requires scan of record to get to n-th field value
- ▶ requires a field for a null value

Variable-Length Records (2)

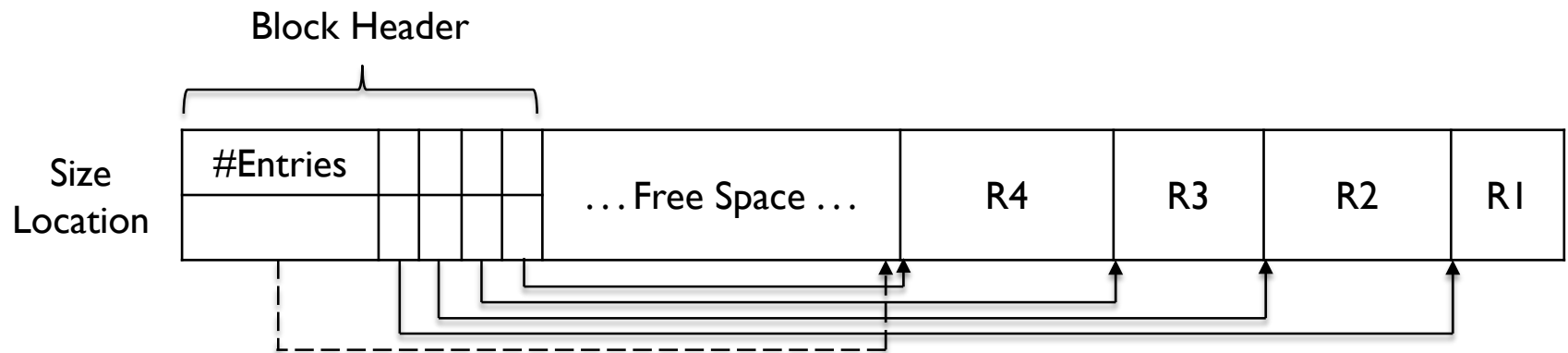
- ▶ Approaches to store variable length records (cont.):
 - ▶ Each record has an *array of field offsets*



- + For the overhead of the offset, we get direct access to any field
 - + Null values: begin/end pointer of a field point to the same address
- ▶ Variable length records typically cause problems in case of (record) attribute modifications:
 - ▶ Growth of a field requires shifting all other fields
 - ▶ A modified record may no longer fit into the block (leave forwarding address at the old location)
 - ▶ A record can span multiple blocks

Block formats for variable length records

- ▶ Each record is identified by a **record identifier (rid)** (or **tuple identifier (tid)**).
- ▶ The rid/tid contains number of block and position in block → simplifies shifting a record from one position/block to another position/block
- ▶ Allocation of records in a block is based on **slotted block structure**:

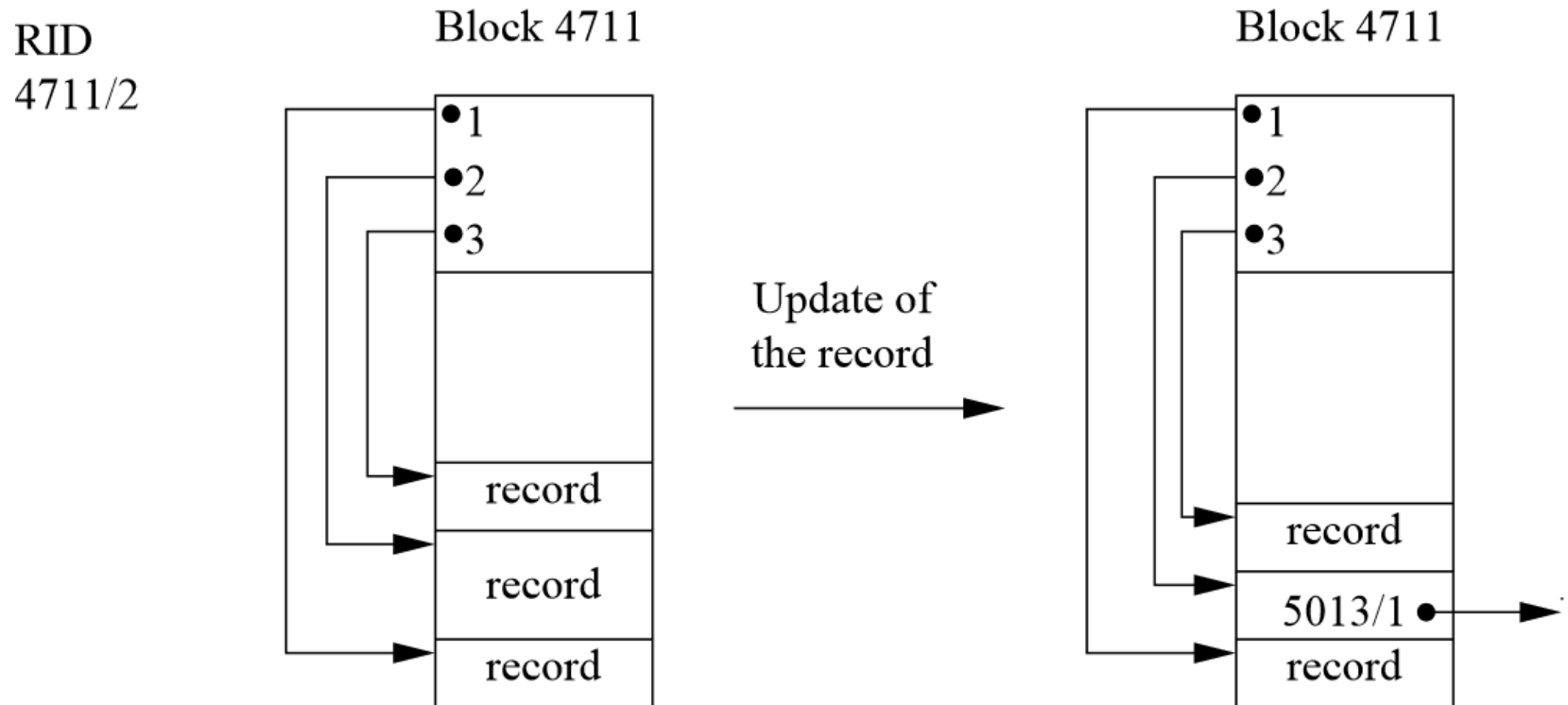


Block formats for variable length records (2)



- ▶ block header contains number of record entries, end of the free space in the block, and location and size of each record
- ▶ records are inserted from the end of the block
- ▶ records can be moved around in block to keep them contiguous

RID-Concept



Organization of Records in Files

Requirements: A file must (efficiently) support

- ▶ insert/delete/update of a record
- ▶ access to a record (typically using rid)
- ▶ scan of all records

Ways to organize blocks (pages) in a file:

- ▶ **Heap File** (unsorted file)
 - ▶ most simple file structure
 - ▶ contains records in no particular order
 - ▶ record can be placed anywhere in the file where there is space
- ▶ **Sequential File**
 - ▶ records are stored in sequential order, based on the value of the search key of each record
- ▶ **Clustered File**
 - ▶ records of different relations are stored in one file
 - ▶ blocks then contain records for more than one relation
- ▶ **Hash File**
 - ▶ the result of a hash function determines in which block of the file the record should be placed

Heap File Organization

- ▶ At database run-time, pages/blocks are allocated and de-allocated
 - ▶ Information to maintain for a heap file includes pages, free space on pages, records on a page
 - ▶ A typical implementation is based on a two doubly linked list of pages; starting with header block.
 - ▶ Two lists can be associated with header block:
 - ▶ full page list
 - ▶ list of pages having free space
- ⇒ Only useful when scanning is the main operation on a file (record can only be accessed efficiently when its **rid** is known)

Heap File

Operations:

- ▶ search
- ▶ insert
- ▶ delete
- ▶ update

Queries:

SELECT Name FROM Employees
WHERE EmpNo = 8832 ;

exact match query

... WHERE EmpNo > 7000 ;

range query

... WHERE DateOfBirth > 01.01.1975 ;

8832	Müller	Max	...	11.11.1973
5588	Sauer	Susi	...	05.10.1960
4777	Fischer	Falk	...	31.10.1958
9999	Abel	Anja	...	10.05.1969

Page 42

6834	Korn	Katja	...	24.09.1984
7754	Opel	Olaf	...	25.02.1976
9912	Ernst	Eric	...	04.04.1970
FREE				

Page 47

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**.
- ▶ Deletions of records are managed using pointer chains.
- ▶ Insertions – must locate the position in the file where the record is to be inserted
 - ▶ if there is free space, insert the record there
 - ▶ if no free space, insert the record in an **overflow block**
 - ▶ in either case, pointer chain must be updated
- ▶ If many record modifications (in particular insertions and deletions), correspondence between search key order and physical order can be totally lost \Rightarrow file reorganization

Sequential File

Operations:

- ▶ search
- ▶ insert
- ▶ delete
- ▶ update

Queries:


SELECT Name FROM Employees
WHERE EmpNo = 8832 ;

exact match query

... WHERE EmpNo > 7000 ;

range query

... WHERE DateOfBirth > 01.01.1975 ;



4777	Fischer	Falk	...	31.10.1958
5588	Sauer	Susi	...	05.10.1960
6834	Korn	Katja	...	24.09.1984
7754	Opel	Olaf	...	25.02.1976

Page 42

8832	Müller	Max	...	11.11.1973
9912	Ernst	Eric	...	04.04.1970
9999	Abel	Anja	...	10.05.1969
FREE				

Page 47

Clustering File Organization

- ▶ Simple file structure stores each relation in a separate file
- ▶ Can instead store several relations in one file using a **Clustering File Organization**; typically based on a **clustering attribute**
- ▶ E.g., clustering organization of the two relations *EMP* and *DEPT*:

CLARK	7782	2450	10	ACCOUNTING
KING	7839	5000		
SMITH	7369	800	20	RESEARCH
ADAMS	7876	1100		
FORD	7902	3000		
ALLEN	7499	1600	30	SALES
BLAKE	7698	2850		

Clustering File Organization (2)

- ▶ Very efficient for queries involving join operator on \bowtie clustering attribute(s), e.g., *EMP* \bowtie *DEPT* (and possibly less efficient for other queries)
- ▶ Results in variable length records
- ▶ Typically used in combination with a cluster index

Indexing: Basic Concepts

- ▶ Indexing mechanisms are used to optimize certain accesses to data (records) managed in files.
 - ▶ For example, the author catalog in a library is a type of index.
- ▶ **Search Key (definition)**: attribute or combination of attributes used to look up records in a file.
- ▶ An **Index File** consists of records (called index entries) of the form

search key value	pointer to block in data file
------------------	-------------------------------
- ▶ Index files are typically much smaller than the original file because only the values for search key and pointer are stored.
- ▶ There are two basic types of indexes:
 - ▶ **Ordered indexes**: Search keys are stored in a sorted order (main focus here in class).
 - ▶ **Hash indexes**: Search keys are distributed uniformly across “buckets” using a hash function.

Index-Sequential File

Page
2

4777	42
8832	47
FREE	

Page
42

4777	Fischer	Falk	...	31.10.1958
5588	Sauer	Susi	...	05.10.1960
6834	Korn	Katja	...	24.09.1984
7754	Opel	Olaf	...	25.02.1976

Page
47

8832	Müller	Max	...	11.11.1973
9912	Ernst	Eric	...	04.04.1970
9999	Abel	Anja	...	10.05.1969
FREE				

Operations:

- ▶ search
- ▶ insert
- ▶ delete
- ▶ update

Queries:

SELECT Name FROM Employees

WHERE EmpNo = 9912 ; exact match query

... WHERE EmpNo > 7000 ; range query

... WHERE DateOfBirth > 01.01.1975 ;

Index Evaluation Criteria

Indexing techniques are evaluated on the basis of:

- ▶ Access types that are efficiently supported; for example,
 - ▶ search for records with specified values for an attribute
(**select * from EMP where EmpNo = 4711**)
 - ▶ search for records with an attribute value in a specified range
(**select from EMP where DeptNo between 20 and 50**)
- ▶ Access time (index entry → record)
- ▶ Insertion time (record → index entry)
- ▶ Deletion time (record → index entry)
- ▶ Space and time overhead (for maintaining index)

Types of Single Level Ordered Indexes

- ▶ In an **ordered index file**, index entries are stored sorted by the search key value.
- ▶ **Primary Index**: in a sequentially ordered file (e.g., for a relation), the index whose search key specifies the sequential order of the file. For a relation, there can be at most one primary index. (→ index-sequential file)
- ▶ **Secondary Index**: an index whose search key is different from the sequential order of the file (i.e., records in the file are not ordered according to secondary index).
- ▶ If search key does not correspond to primary key (of a relation), then multiple records can have the same search key value → **clustering index**

Types of Single Level Ordered Indexes (2)

- ▶ **Dense Index Files:** index entry appears for every search key value in the record file.
- ▶ **Sparse Index Files:** only index entries for some search key values are recorded.
 - ▶ To locate a record with search key value K , first find index entry with largest search key value $< K$, then search file sequentially starting at the record the index entry points to
 - ▶ Less space and maintenance overhead for insertions and deletions
 - ▶ Generally slower than dense index for directly locating records

Secondary Indexes

- ▶ Often one wants to find all records whose values in a certain field (which is **not** the search key of the primary index) satisfy some condition
 - ▶ Example 1: In the employee database, records are stored sequentially by EmpNo, we want to find employees working in a particular department.
 - ▶ Example 2: as above, but we want to find all employees with a specified salary or range of salary
- ▶ One can specify a secondary index with an index entry for each search key value; index entry points to a **bucket**, which contains pointers to all the actual records with that particular search key.

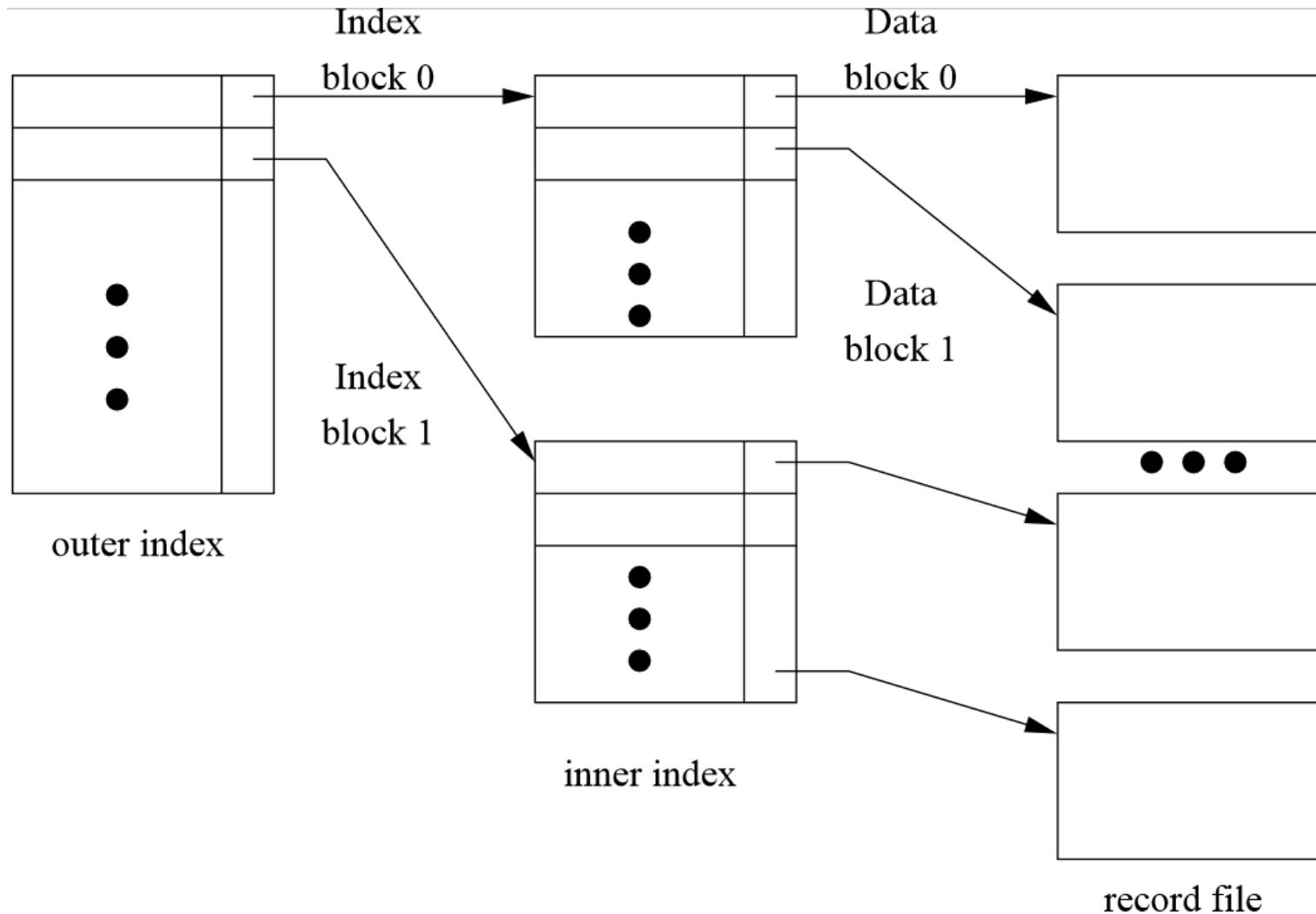
Primary Indexes vs. Secondary Indexes

- ▶ Secondary indexes have to be dense
- ▶ Indexes offer substantial benefits when searching for records
- ▶ When a record file is modified (e.g., a relation), every index on that file must be updated. Updating indexes imposes overhead on database performance.
- ▶ 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)

Multi-Level Index

- ▶ If primary index does not fit in memory, access to records becomes expensive
- ▶ To reduce number of disk accesses to index entries, treat primary index on disk as sequential file and construct a sparse index on it.
 - outer index → a sparse index of primary index
 - inner index → the primary index file
- ▶ If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- ▶ Note that indexes at all levels must be updated on insertions or deletions of records from a file.

Multi-Level Index Structure



Dynamic Multi-Level Indexes using B⁺-Trees

B⁺-Tree indexes are an alternative to index sequential files.

- ▶ Disadvantage of index-sequential files:
 - ▶ performance degrades as sequential file grows, because many overflow blocks are created.
 - ▶ Periodic reorganization of entire file is required.
- ▶ Advantage of B⁺-Tree index file:
 - ▶ automatically reorganizes itself with small, local changes in the case of insertions and deletions.
 - ▶ Reorganization of entire file is not required to maintain performance.
- ▶ Disadvantage of B⁺-Trees:
 - ▶ extra insertions and deletion overhead, space overhead.
- ▶ Advantages of B⁺-Trees outweigh disadvantages, and B⁺-Trees are used extensively in all DBMS.

Dynamic Multi-Level Indexes using B⁺-Trees (2)

A B⁺-Tree is a rooted tree satisfying the following properties:

- ▶ All paths from the root to leaf have the same length (\Rightarrow a B⁺-Tree is a balanced tree).
- ▶ Each node that is not the root or a leaf node has between $\lceil n/2 \rceil$ and n children (where n is fixed for a particular tree).
- ▶ 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, it can have between 0 and $n - 1$ values.

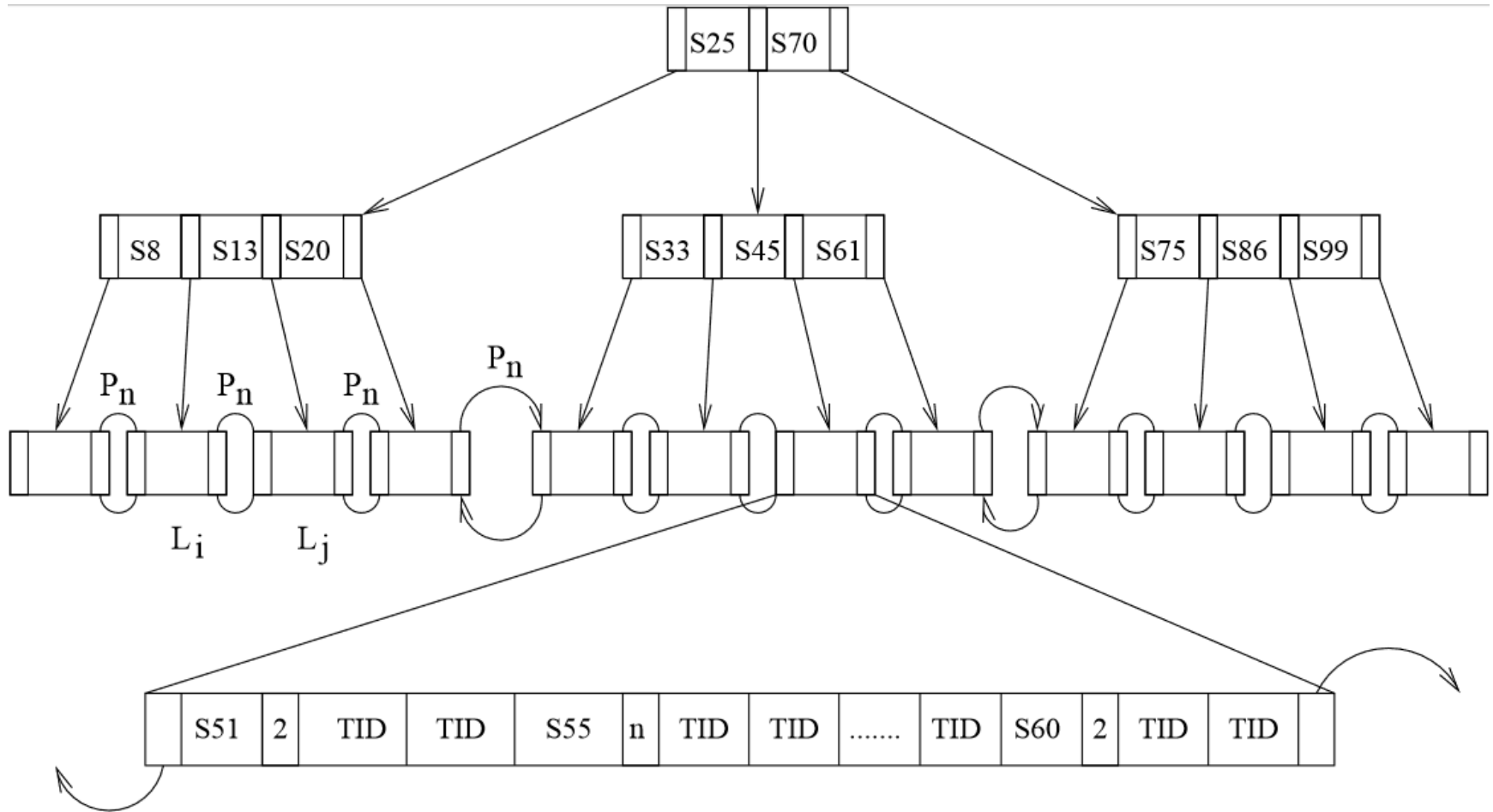
Dynamic Multi-Level Indexes using B⁺-Trees (3)

- ▶ Typical structure of a node:

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- ▶ K_i are the search key values
- ▶ P_i 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, i.e.,
 $K_1 < K_2 < K_3 \dots < K_{n-1}$

Example of a B⁺-Tree



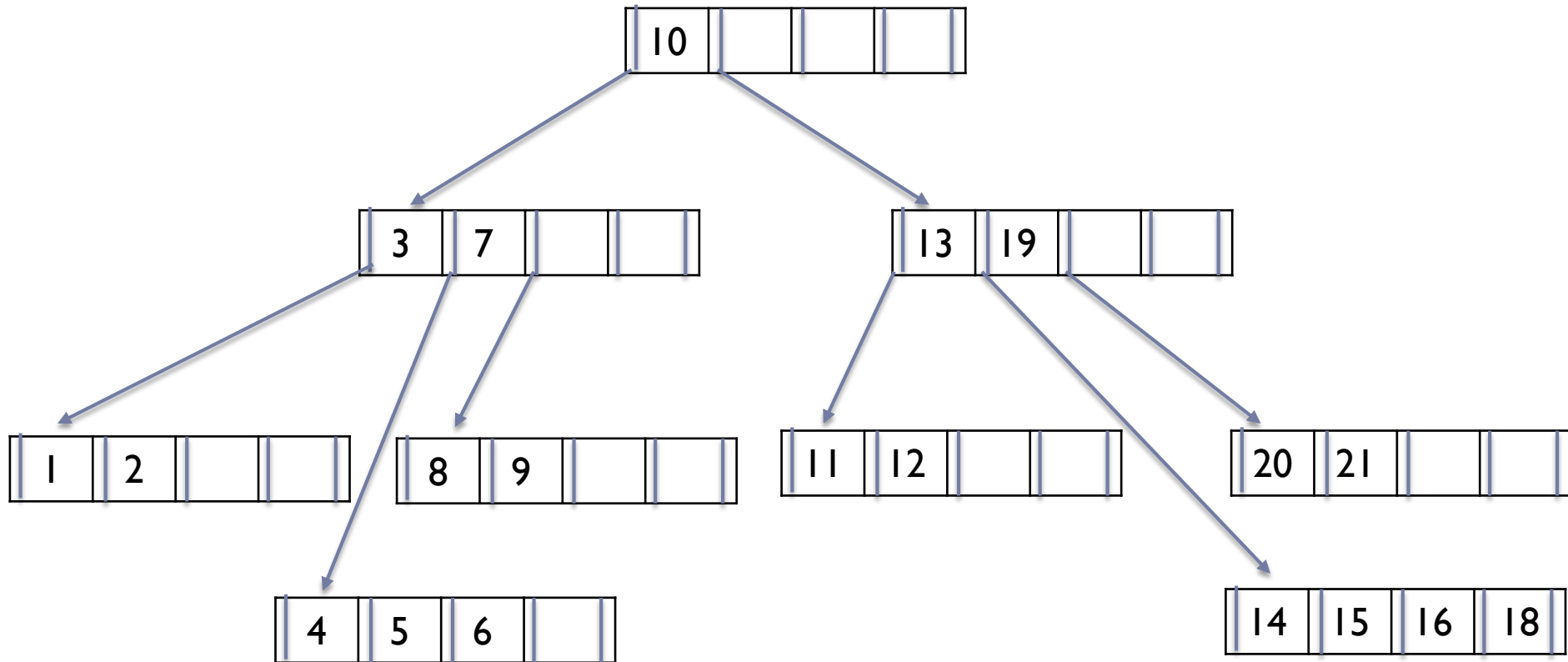
Leaf Nodes in a B⁺-Tree

- ▶ For $i = 1, 2, \dots, n - 1$, pointer P_i either points to a file record with search key value K_i (using the tuple identifier – tid), or to a bucket of pointers to file records, each record having search key value K_i .
 - ▶ Note that one only needs bucket structure if search key does not correspond to primary key of relation the index is associated with.
- ▶ If L_i, L_j are leaf nodes and $i < j$, L_i 's search key values are less than L_j 's search key values.
- ▶ P_n points to next leaf node in search key order.

Non-Leaf Nodes in a B⁺-Tree

- ▶ All the search keys in the subtree to which P_1 points are less than K_1
- ▶ All search keys in the subtree to which P_m points are greater than or equal to K_{m-1} .

B⁺-Tree



Observations about B⁺-Trees

- ▶ Since the inter-node connections are done by pointers, there is no assumption that in the B⁺-Tree logically close blocks are also “physically” close.
- ▶ The non-leaf levels of the B⁺-Tree form a hierarchy of sparse indices.
- ▶ The B⁺-Tree contains a relatively small number of levels (logarithmic in size of the main file), thus searches can be done efficiently.
- ▶ 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

- ▶ Start with the root node
 - ▶ Examine the node for the smallest search key value $> k$.
 - ▶ If such a value exists, assume it is K_i . Then follow P_i to the child node.
 - ▶ Otherwise, $k \geq K_{m-1}$, where m pointers in the node. Then follow P_m to the child node.
- ▶ If the node is reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
- ▶ Eventually reach a leaf node. Scan entries K_i in the leaf node. If $K_i = k$, follow pointer P_i to the desired record or bucket. Otherwise no record with search key value k exists.

Queries on B⁺-Trees: Further Comments

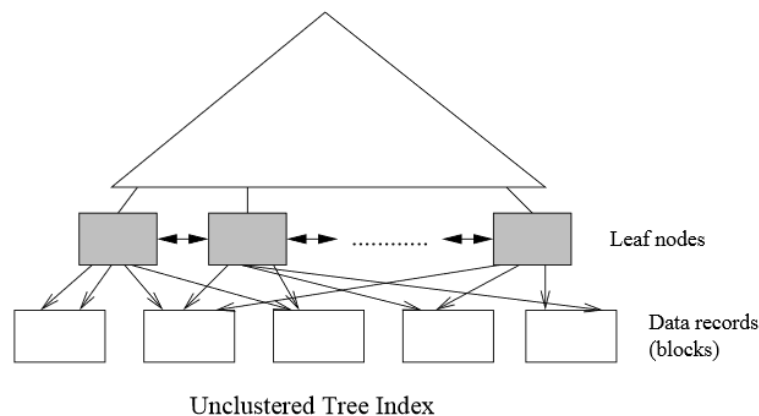
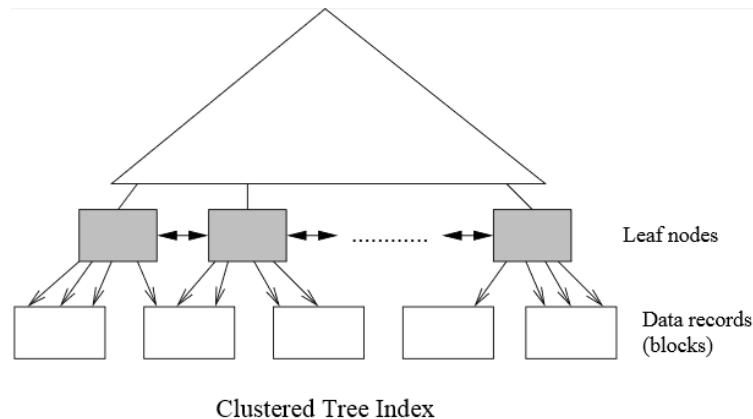
- ▶ If there are V search key values in the file, the path from the root to a leaf node is no longer than $\lceil \log_{[n/2]}(V) \rceil$.
- ▶ In general a node has the same size as a disk block, typically 4KB, and $n \approx 100$ (40 bytes per index entry).
- ▶ With 1,000,000 search key values and $n = 100$, at most $\log_{50}(1,000,000) = 4$ nodes are accessed in the lookup!

B⁺-Tree File Organisation

- ▶ Index file degradation problem is solved by using B⁺-Tree indices.
- ▶ Data file degradation problem is solved by using a B⁺-Tree file organization.
- ▶ Leaf nodes in a B⁺-Tree file organization can store records instead of just pointers.

Clustered vs. Unclustered Indices

Clustered: Order of data records is the same as order of index entries.



Index Definition in SQL

- ▶ Syntax (most general):

```
create [unique] index ⟨index name⟩ on  
    ⟨relation name⟩ ( ⟨list of attributes⟩ );  
  
drop index ⟨index name⟩ ;
```

- ▶ Many more options available, including clauses to specify bitmaps, tablespace, cluster, key compression, function-based index, (hash) partitioned index, ...
- ▶ By default, indexes are created in ascending order.
- ▶ With primary key in a relation, an index is associated.
- ▶ Example:

```
create index city_name_idx on CITY ( name );
```

Summary

- ▶ Storage hierarchy
- ▶ Blocks formats
- ▶ Record identifiers
- ▶ Buffer management
- ▶ B+-Tree