ĐẠI HỌC
BÁCH KHOA

25 YEARS ANNIVERSARY
SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3090E - Databases

Chapter 8: Indexing

Muriel VISANI

murielv@soict.hust.edu.vn

Outline

- Overview of database storage structures
- Physical database files
- Database index

Objectives

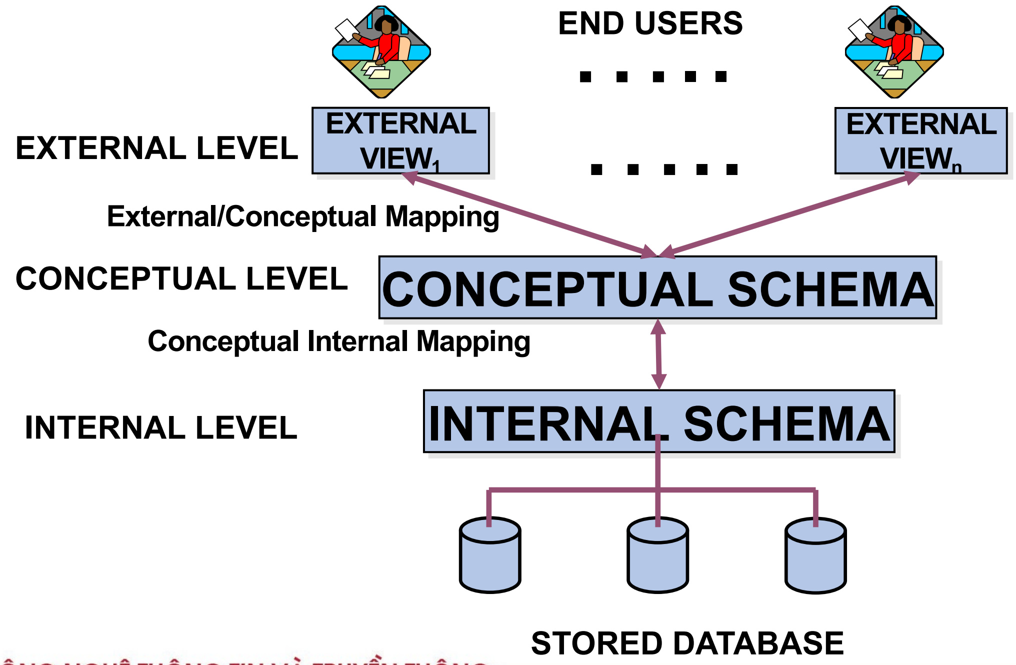
- Upon completion of this lesson, students will be able to:
  - Understand the physical database files
  - Understand the role of database indexes

Keywords

Heap file	Files of Unordered Records
Ordered file	Physically order the records of a file on disk based on the values of one of their fields (key field)
Index	A data structure that improves the speed of data retrieval operations
B-tree	A self-balancing tree data structure that enables efficient search

1. Overview of database storage structures

- ***3-tier Schema Model (ANSI-SPARC Architecture)***



1. Overview of database storage structures

- ***How does Mariadb store data***

```
MariaDB [(none)]> SHOW VARIABLES LIKE 'datadir';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| datadir       | /var/lib/mysql/ |
+---------------+-----------------+
```

```
MariaDB [student_management]> show tables;
+------------------------------+
| Tables_in_student_management |
+------------------------------+
| class                        |
| enrolled                     |
| faculty                      |
| student                      |
+------------------------------+
```

```
:/var/lib/mysql/student_management# ls -la

ql mysql   4096 Mar 12 02:05 .
ql mysql   4096 May  5 06:06 ..
ql mysql   1547 Mar 12 02:05 class.frm
ql mysql 114688 Mar 12 02:21 class.ibd
ql mysql     65 Mar 12 01:59 db.opt
ql mysql   1466 Mar 12 02:03 enrolled.frm
ql mysql 114688 Mar 12 02:18 enrolled.ibd
ql mysql   1005 Mar 12 02:04 faculty.frm
ql mysql  98304 Mar 12 02:16 faculty.ibd
ql mysql   1101 Mar 12 02:00 student.frm
ql mysql  98304 Mar 12 02:23 student.ibd
```

the .frm table file stores the table's format

the .ibd file stores the table's data

1. Overview of database storage structures

- ***How does Mariadb store data***
  - the .frm file stores the table's format

```
MariaDB [student_management]> describe student;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| snum  | int(11)     | NO   | PRI | NULL    |       |
| sname | varchar(40) | YES  |     | NULL    |       |
| major | varchar(30) | YES  |     | NULL    |       |
| level | varchar(10) | YES  |     | NULL    |       |
| age   | int(11)     | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

```
root@285e07e9458f:/var/lib/mysql/student_management# cat student.frm
?

VM?\!   ?s?$??%?牮B??
??PRIMARY??InnoDB??f\P
                      (/?
N?
 ?snum?sname?major?level?age?root@285e07e9458f:/var/lib/mysql/student
```

1. Overview of database storage structures

- ***How does Mariadb store data***
  - the .ibd file stores the table's data

```
MariaDB [student_management]> select * from student;
+------+------------------+---------+-------+-----+
| snum | sname            | major   | level | age |
+------+------------------+---------+-------+-----+
|    1 | Nguyen Van A     | CS      | JR    |  18 |
|    2 | Nguyen Viet Cuong| History | JR    |  19 |
|    3 | Nguyen Hong Ngoc | CS      | JR    |  19 |
|    4 | Mark Juke        | History | JR    |  20 |
|    5 | Elon Mulk        | CS      | JR    |  20 |
|    6 | Donal Trump      | CS      | JR    |  20 |
|    7 | Obama            | CS      | JR    |  20 |
|    8 | Tan Dung         | History | SR    |  30 |
+------+------------------+---------+-------+-----+
```

```
root@285e07e9458f:/var/lib/mysql/student_management# cat student.ibd
???]&!??????????????????????????&&???????????????????????????]&??Y?&??Y?&???j?&???
?????????????????????????????i??????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????????????????????
????????????i???????????????????????????????????????????????????????????????????????
?????????????????????????????????????????????????????????????j?&??Q?????????'??E?
9infimum
        supremum
                .?8?WNguyen Van ACSJR?8?:?cNguyen Viet CuongHistoryJR?  2?@??Nguyen H
ong NgocCSJR?    (0?1?iMark JukeHistoryJR?       0+?0??Elon MulkCSJR?
                                                                    8-?Q?kDonal Trum
pCSJR?@'?W??ObamaCSJRH?????Tan DungHistorySR?pc??Q?'??root@285e07e9458f:/var/lib/mys
```

VIỆN CÔNG

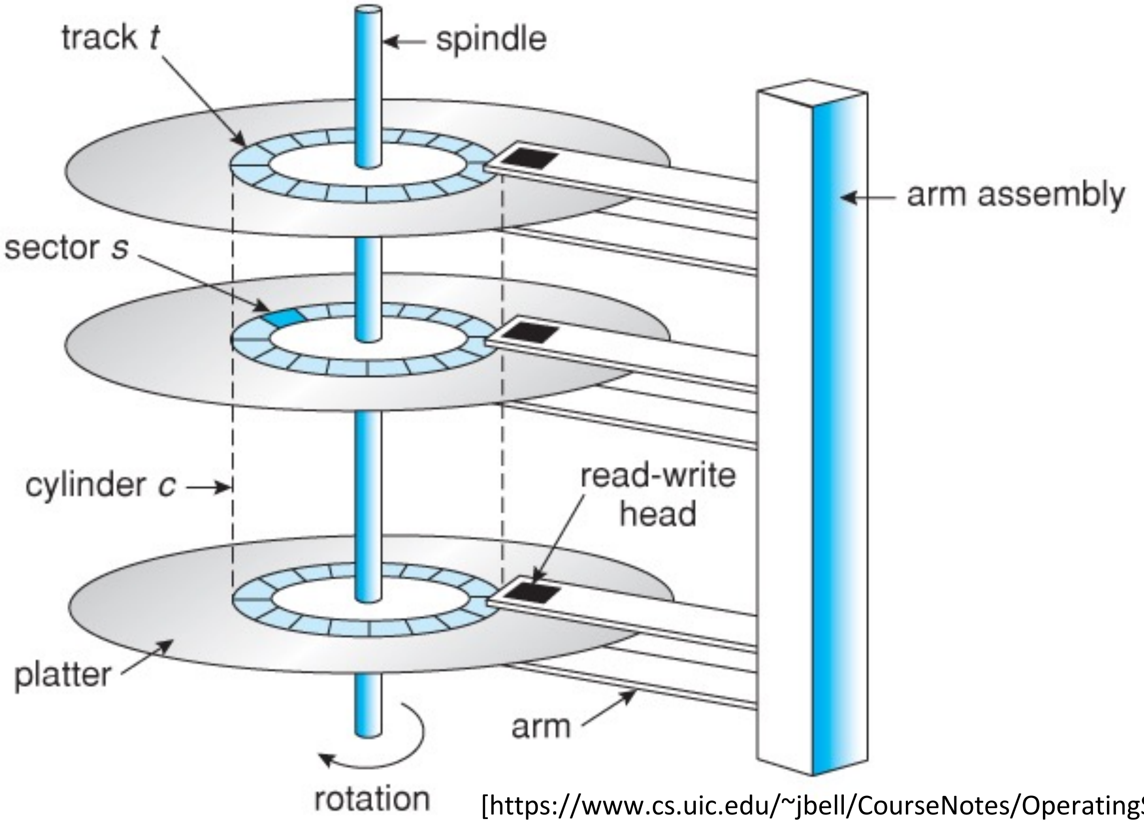
2. Physical database files

- Motivation
- Magnetic disks as data storage
- Primary file organizations

2.1. Motivation

- Databases typically store large amounts of data persistently on disks, because:
  - Databases are too large to fit entirely in main memory (RAM), but they can fit entirely on the disk
  - Disk storage is non-volatile, whereas main memory is volatile
- But:
  - Disk access is extremely slow, compared to CPU (processing) speed
  - So, the performance of the DBMS largely depends on the number of disk I/O operations that must be performed

2.2. Magnetic disks as data storage



[https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/10_MassStorage.html]

2.2. Magnetic disks as data storage

- A disk is a random access addressable device.
- Data is transferred between disk and main memory in units called **blocks**.
  - A block is a contiguous sequence of bytes from a single **track** of one platter.
  - Typical disk block sizes: 4KB – 8KB.
- Disk I/O (read/write from disk to main memory) **overhead** is the key factor of database performance optimization.
  - Disk overhead: the disk space required for information that is not data, but is used for location and timing

2.2. Magnetic disks as data storage

- Data files are decomposed into **pages**
  - Fixed size piece of contiguous information in the file
  - Unit of exchange between disk and main memory

- The disk is divided into blocks which have same size as pages
  - So that a page can be stored in any block

- Application's request for **read** item satisfied by:
  - Read page containing item to buffer in DBMS
  - Transfer item from buffer to application

- Application's request to **update** item satisfied by:
  - Read page containing item to buffer in DBMS (if it is not already there)
  - Update item in DBMS (main memory) buffer
  - Copy buffer page to page on disk

2.2. Magnetic disks as data storage

- I/O access to a page – **time** needed
  - Seek latency: time to position heads over cylinder containing page (some ms)
  - Rotational latency: additional time for platters to rotate so that start of block containing page is under head (some ms)
  - Transfer time: time for platter to rotate over block containing page (depends on block size)
  - Latency = seek latency + rotational latency

- Our goal is to minimize the average latency, and reduce the number of page transfers
  - Therefore, try to store pages containing related information close together on disk

  => Data file **organization**

2.2.1. Physical database design

- The process of physical database design involves choosing the particular data organization techniques that best suits the given application requirements
  - on SELECT, INSERT, UPDATE, DELETE

- The data stored on disk is organized as files of records:
  - Primary file organizations: determine how the file records are physically placed on the disk, and hence how the records can be accessed.
  - Secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields.

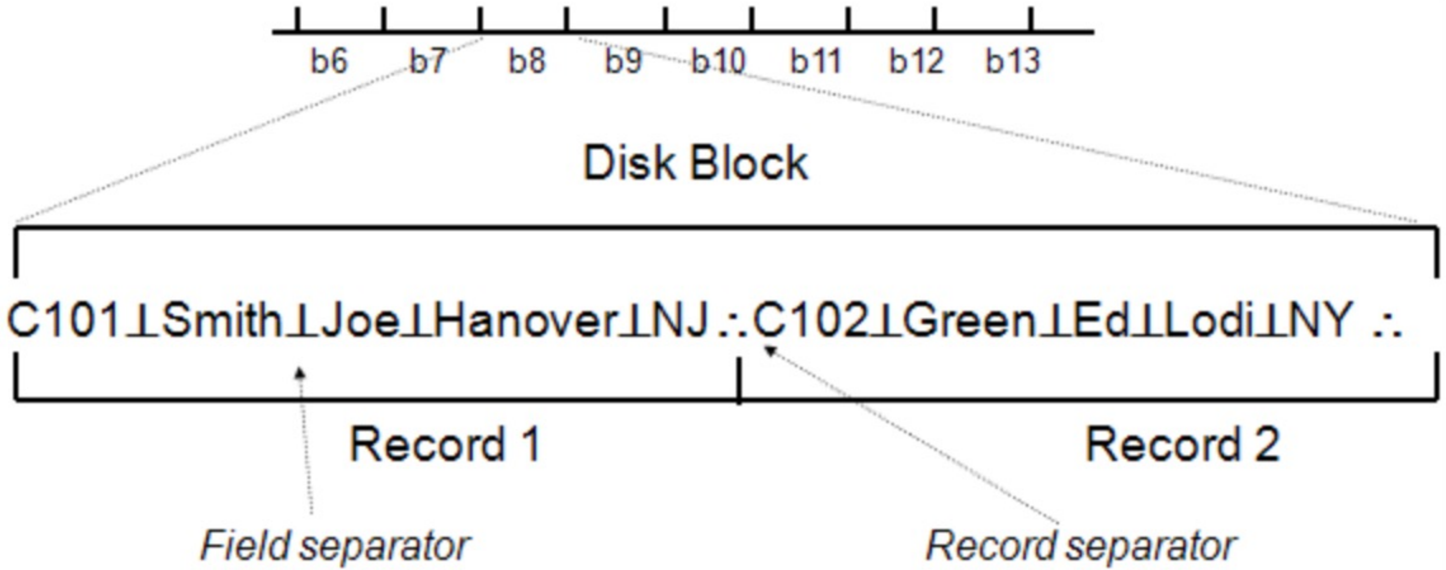
2.2.1. Physical database design

- **Primary file organizations** determine how the records of a file are physically placed on the disk, and hence how the records can be accessed.
  - A **heap file** (or **unordered file**) places the records on disk in no particular order by appending new records at the end of the file.
  - A **sorted file** (or **sequential file**) keeps the records ordered by the value of a particular field (called the sort key).
  - A **hashed file** uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk.
  - **N.B.:** The terminology may carry different meaning in different context. Sequential files in COBOL , for example, are unordered files.

2.2.1. Physical database design

- A **secondary organization** or **auxiliary access structure** allows efficient access to the records of a file based on alternate fields than those that have been used for the primary file organization.
  - Most of these exist as **indexes.**

2.2.2. Placing File Records on Disk



[http://holowczak.com/database-record-storage-index-structures/?doing_wp_cron=1622020116.6039350032806396484375]

2.2.2. Placing File Records on Disk

- Some definitions:
  - **Fixed length** records: Each record is of fixed length. Pad with spaces in each field
  - **Variable Length** records: Each record is as long as the data it contains.
  - **Unspanned** Records: A record is found in one and only one block. i.e., records do not span across block boundaries.
  - **Spanned** Records: Records are allowed to span across block boundaries.

- Often, to avoid too many disk I/O access, we try to avoid spanned records
  - **Blocking Factor:** number of tuples (records) that can fit into a single block.
    - Easier to set when there is no variable length attributes (*e.g.* type text (varchar with no upper limit))
    - Example: EMPLOYEE takes 100 bytes to store one tuple (record).
      - If the Block Size is 2,000 bytes, then we can store 20 EMPLOYEE tuples (records) in one block. Thus the *Blocking factor* is 2000/100 = 20

2.3. Primary file organizations

- More details about:
  - Files of Unordered Records (Heap Files)
  - Files of Ordered Records (Sorted Files)
  - Hashing Techniques

2.3. Primary file organizations

- **Files of Unordered Records (Heap Files)**
  - Records are placed in the file in the order in which they are inserted
  - INSERT: Inserting a new record is very efficient
    - New records are inserted at the end of the file
    - Insert takes constant time.
  - DELETE: not efficient
    - Delete takes o(n/2) time to locate the record to delete, then constant time to actually delete it
    - Leaves unused space in the disk block
      - require periodic reorganization
  - SELECT / UPDATE: not efficient
    - Select takes o(n/2) time (*n* is the number of records)
    - Update take o(n/2) time to locate the record to be updated + constant time (similar to Delete followed by Insert).

2.3. Primary file organizations

- **Files of Ordered Records (Sorted Files)**

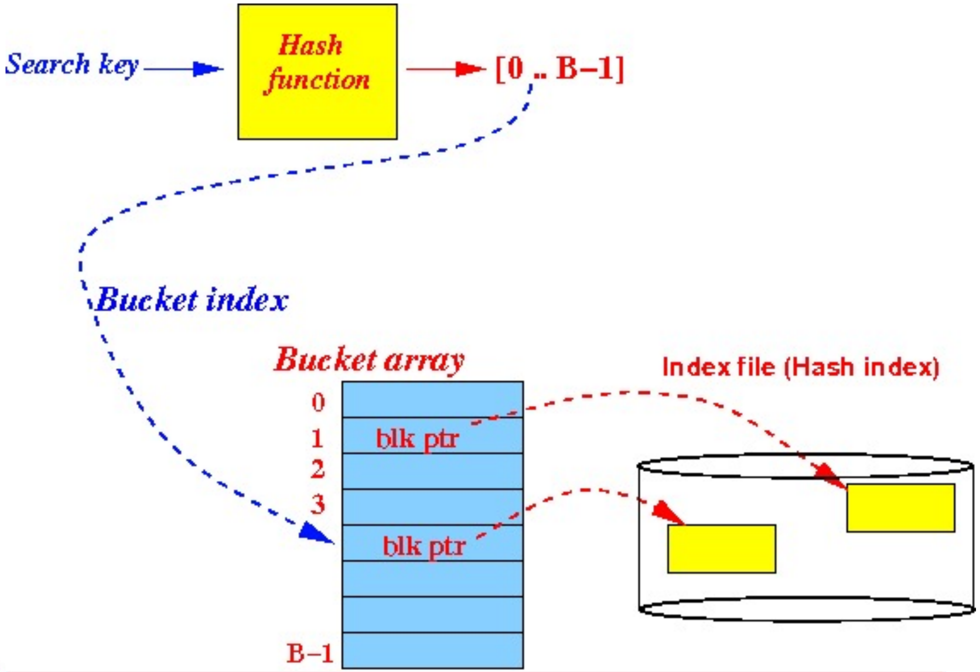
  - Physically order the records of a file on disk based on the values of one of their fields (key field)

  - SELECT: binary search (very fast)
    - Select takes $o(\log_2 n)$

  - INSERT/DELETE/UPDATE: more expensive
    - Insert (resp. delete) takes $o(\log_2 n)$ time to find the location for the new (resp. old) record plus o(n/2) to re-organize records
    - Update takes $o(\log_2 n)$ to locate the record to be updated, then another $o(\log_2 n)$ to locate the new location plus another o(n/2) to re-organize the rest of the data.

2.3. Primary file organizations

- **Hash files**
  - The address of the disk block in which the record is stored is the result of applying a hash function to the value of a particular field (hash field) of the record.
  - Very fast access to records for search on equality condition on the hash field.



[Emory Computer Science]

2.3. Primary file organizations

- **Hash files**
  - **Advantages**
    - If the hash field value is exactly known, then SELECT, INSERT, UPDATE, DELETE are very quick
    - Multiple records can be accessed at the same time (the storage location is independent between different records)
    - Suitable for online transaction systems like online banking, ticket booking system etc.

2.3. Primary file organizations

- **Hash files**
  - **Disadvantages**
    - If hash field is not unique, can lead to loss of data (older record overwritten by newer record with same hash field value)
    - All the records are randomly scattered in the memory => not efficient memory use
    - If the SELECT/UPDATE/DELETE is not based on the exact hash field(s), then this method is **very inefficient** (exhaustive search)
      - If there is multiple hash columns (*e.g.* name and phone number), and if we are searching any record using phone or name alone, it will not give correct results
    - If we are searching for a range of data, then this method is **very inefficient**
    - Range addresses are independent of the hash field range
      - Example: searching for the students born before 1995 will not be efficient
    - If these hash column(s) are frequently updated, then the data block address is also changed accordingly. Each update will generate new address. This is not acceptable

2.3. Primary file organizations

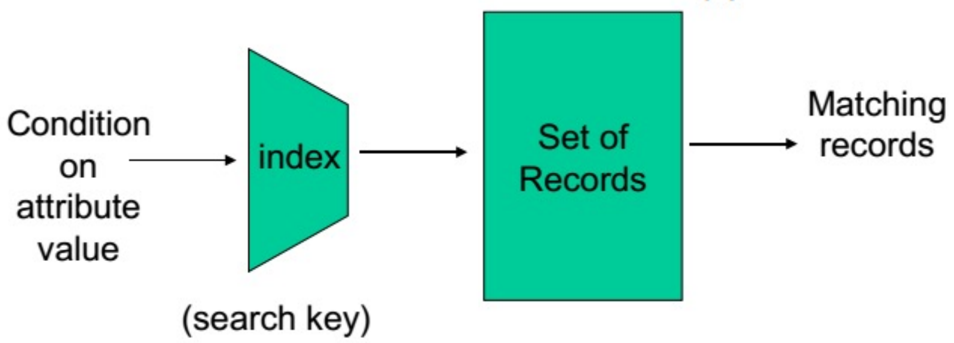
- **Hash files**
  - **Conclusion**
    - **Only** adapted for very specific applications where the data is only accessed by querying a given field (*e.g.* booking number)
    - For other cases, we'll use **secondary file organization** (mostly **indexes**)

3. Database indexes

- What is database index?
- Index data structures
- B+tree
- Sparse vs. Dense index
- Clustered vs. Non-clustered index
- Index creation in SQL
- Using indexes

3.1. What is database index?

- Secondary organization or auxiliary access structure (commonly index) allows efficient access to file records based on alternate fields

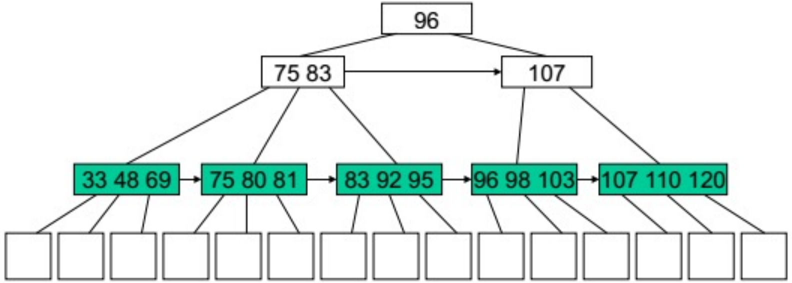


3.2. Index data structures

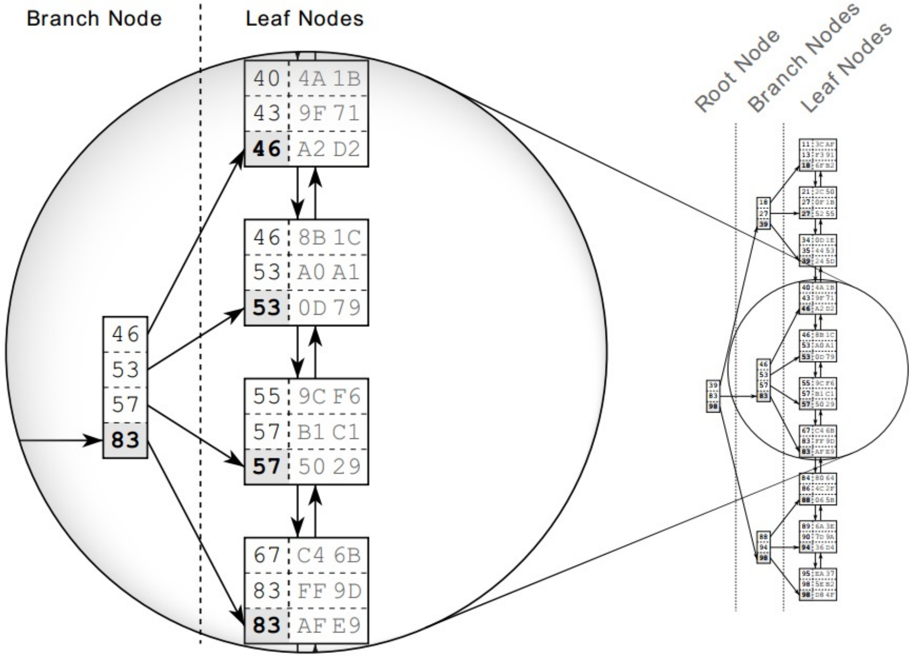
- Indexes can be implemented using different data structures.
  - B+-tree index
  - Traditional hash index
  - Dynamic hash index: number of buckets modified dynamically
  - Bitmap index (more often used in data warehouses than in tradiational relational databases)
  - R-tree: index for specific data types (points, lines, shapes)
  - quadtree: recursively partition a 2D plane into four quadrants
  - octree: quadtree version for three dimensional data
  - main memory indexes: T-tree, binary search tree

3.3. B+Tree

- Balanced tree of key-pointer pairs
- Keys are sorted by value
- Nodes are at least half full
- Access records for key: traverse tree from root to leaf



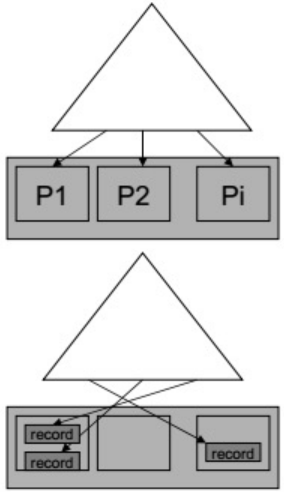
3.3.1. Example: B+ tree



© Gulutzan, Peter, and Trudy Pelzer. *SQL Performance Tuning*. Addison-Wesley Professional, 2003.

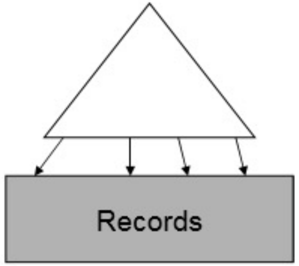
3.4. Sparse vs. Dense index

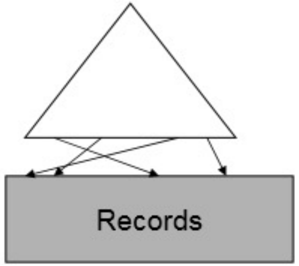
- Sparse index
  - pointers to disk **pages**
  - at most one pointer per disk page
  - usually much less pointers than records
- Dense index
  - pointers to individual **records**
  - one key per record
  - usually more keys than sparse index optimization: store repeating keys only once, followed by pointers



3.5. Clustered vs. Non-Clustered

- Clustered index on attribute X
  - This index controls the placement of records on disk
  - Only one clustered index per table
  - Can be dense or sparse (often sparse)
- Non-clustered index on attribute X
  - No constraint on table organization
  - Can have more than one index per table
  - Always dense

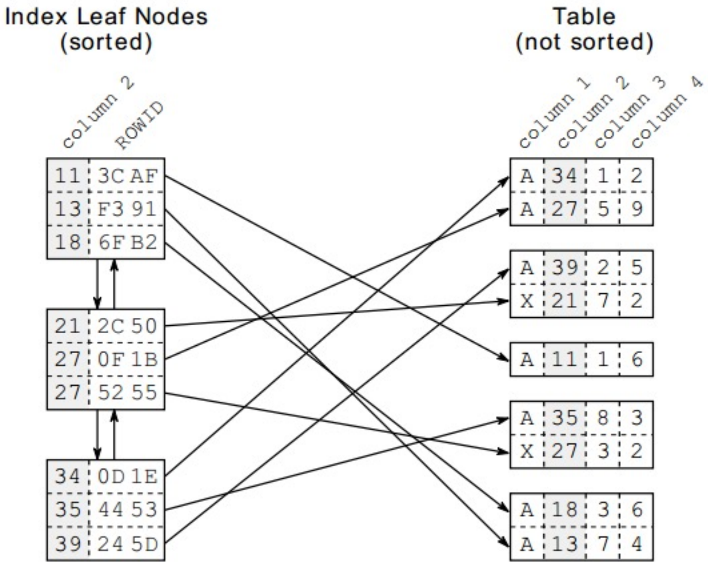




3.5. Clustered vs. Non-Clustered

- More details on clustered *vs.* non-clustered indexes:
  - https://www.geeksforgeeks.org/difference-between-clustered-and-non-clustered-index/
  - **Recall from Chapter 7**: by default, the index for a PK attribute is clustered, wheread the index for a unique attribute is unclustered.

3.5.1. Example: Non-clustered index



© Gulutzan, Peter, and Trudy Pelzer. *SQL Performance Tuning*. Addison-Wesley Professional, 2003.

3.6. Creating Index

- Important remark:
  - All relational DBMS **automatically** index the primary key of each table with a clustered index (B-tree in general)
  - So, if you declare an attribute as PK for a given table, you **DO NOT NEED** to create the corresponding index
  - When you make a search / join based on the Primary Key, you **DO NOT NEED** to specify that the index should be used
    - Because in clustered index, the index **is** the main data

3.6. Creating Index

- Syntax for creating the indexes together with the tables (example with MariaDB) – for non PK attributes

```
CREATE TABLE table_name(
    column_list,
    ...,
    FULLTEXT (column1,column2,..)
);
```

3.6. Creating Index

- Syntax for creating the indexes separately from the tables (with MariaDB) – for non PK attributes

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name [index_type]
ON tbl_name (index_col_name,...) [index_option]
[algorithm_option | lock_option] ...
```

- index_type: USING {BTREE | HASH}
- index_col_name: col_name [(length)] [ASC | DESC]
- index_option: [KEY_BLOCK_SIZE [=] value
{{{|}}} index_type
{{{|}}} WITH PARSER parser_name
{{{|}}} COMMENT 'string’
{{{|}}} CLUSTERING={YES| NO}] [IGNORED | NOT IGNORED]
- algorithm_option: ALGORITHM [=] {DEFAULT|INPLACE|COPY|NOCOPY|INSTANT}
- lock_option: LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}

More details on [https://mariadb.com/kb/en/create-index/]

3.6. Creating Index

- Syntax for dropping indexes
  - The keyword is DROP
  - The syntax differ on the DBMS (some mention the source table, some others don't) => see the documentation for your DBMS

3.7. Using Indexes

- Example: FULLTEXT indexes, using mySQL

```
CREATE TABLE tutorial ( id INT UNSIGNED
AUTO_INCREMENT NOT NULL PRIMARY KEY, title
VARCHAR(200), description TEXT,
FULLTEXT(title,description) ) ENGINE=InnoDB;
```

```
SELECT * FROM tutorial WHERE
MATCH(title,description) AGAINST ('left right' IN
NATURAL LANGUAGE MODE);
```

id	title	description
5	SQL Full Outer Join	In SQL the FULL OUTER JOIN combines the results of both left and right outer joins and
3	SQL Left Join	The SQL LEFT JOIN, joins two tables and fetches rows based on a condition, which are m

[https://www.w3resource.com/mysql/mysql-full-text-search-functions.php]

Summary

- Overview of database storage structures
  - 3-tier Schema Model (ANSI-SPARC Architecture)
  - How MariaDB stores data
- Physical database file structures
  - Motivation
  - Magnetic disks as data storage
  - Primary file organizations
- Database index
  - What is database indexes?
  - Index data structures
  - B+tree
  - Spare vs. Dense index
  - Clustered vs. Non-clustered index
  - Index creation in SQL
  - Using indexes

Exercise 1

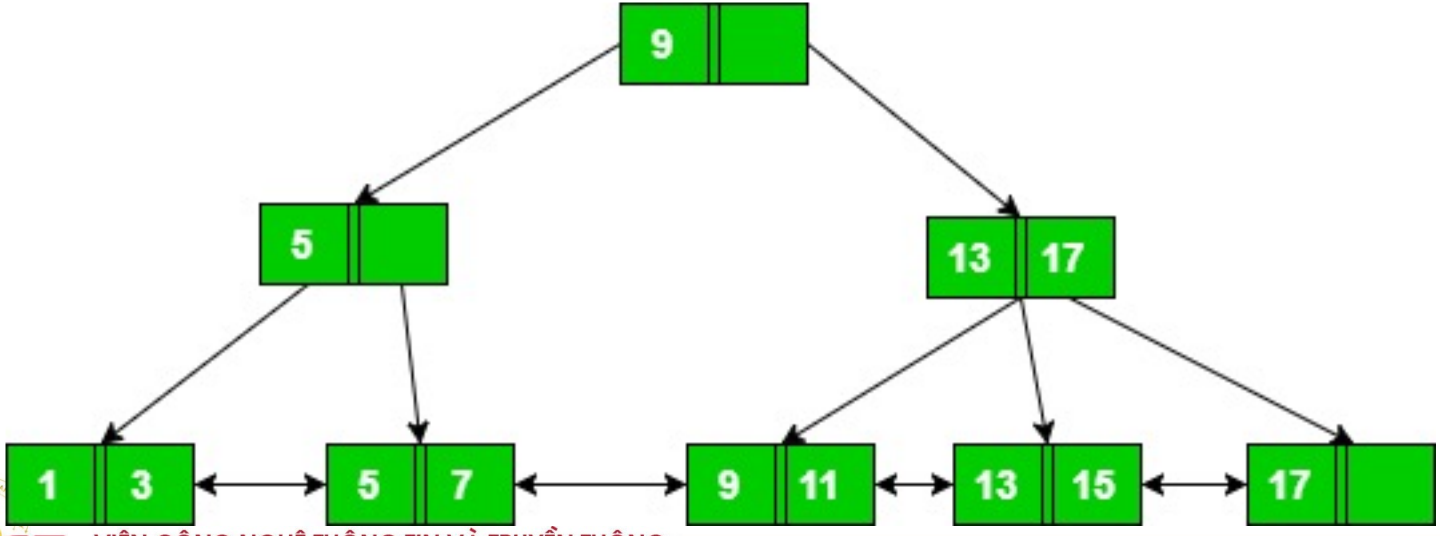
- Read the following definition of B+ tree: https://en.wikipedia.org/wiki/B%2B_tree
  - Extract (with branching factor B):

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b = 7$	Example $b = 100$
Root Node (when it is the only node in the tree)	Records	0	$b - 1$	0–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil$	b	4–7	50–100

- Make the following exercise (with b=4):
  - https://opendsa-server.cs.vt.edu/embed/bPlusTreeInsertPRO#:~:text=In%20this%20exercise%20your%20job,the%20values%20in%20the%20stack.
  - There is a small bug in this API. You might need to run it multiple times to find it. Will you??

Exercise 2

- With the B+ tree (b=2) shown below, how much is the minimum number of nodes (including the root node) that must be fetched in order to satisfy the following query: “Get all records with a search key greater than or equal to 7 and less than 15”



Exercise 2

- Solution:
- Conclusion: B+ trees are very efficient for range searches.

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict