

# Principles of Database Systems (CS307)

## Lecture 13 - 1: Indexing

Zhong-Qiu Wang

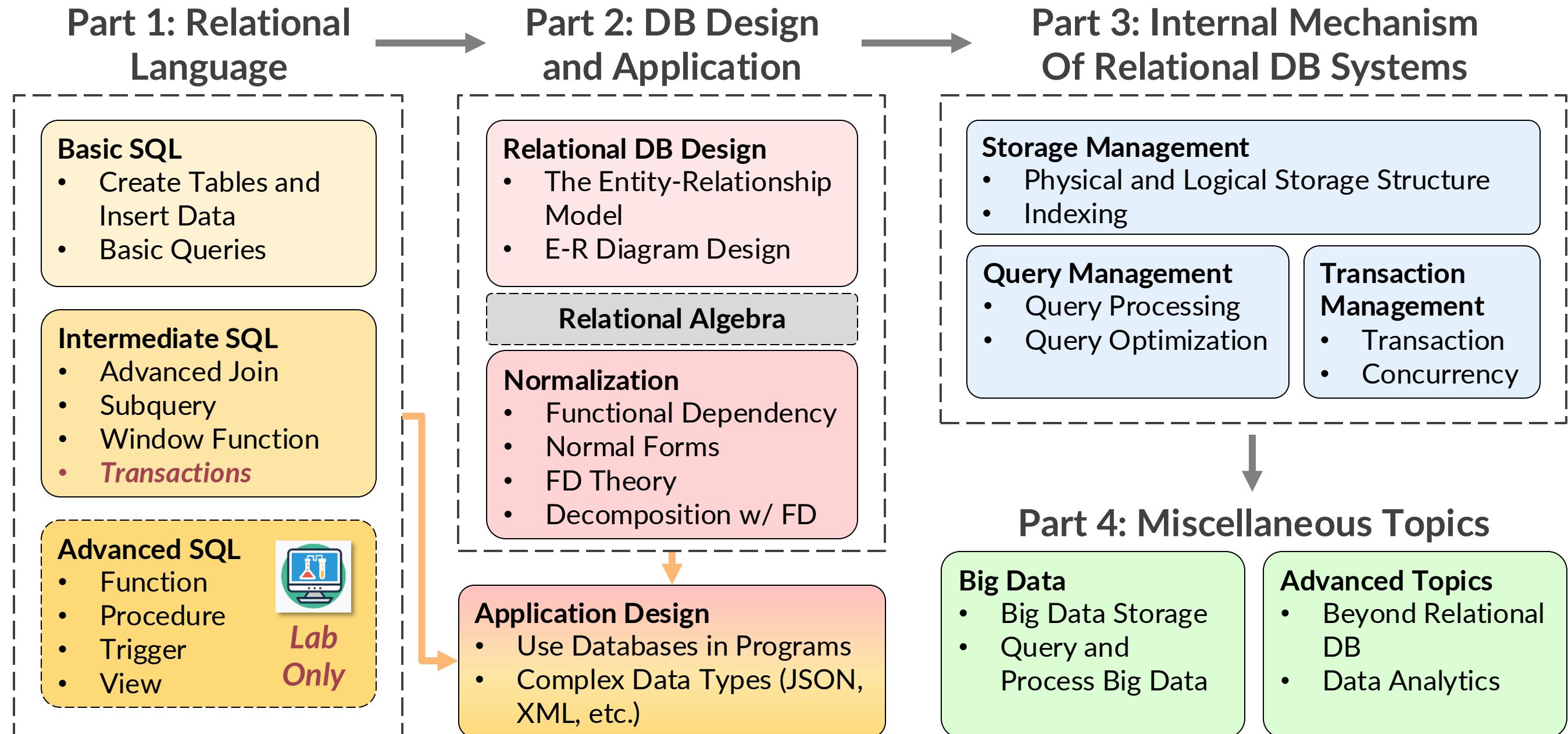
Department of Computer Science and Engineering  
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7<sup>th</sup> Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

# Announcements

- Assignment on Relation Algebra, due date: 22:30 on December 2nd 2025,  
**Beijing Time**
  - Please do not miss the deadline
  - On blackboard, **the assignment is in the --Tests tab, not in the -Assignments tab**

# Outline



# Introduction

# Motivation

- Many queries reference only a small proportion of the records in a file
  - E.g., find all instructors in the Physics department
    - Inefficient to read every tuple in the instructor relation to check if the dept name value is “Physics”
    - The system should be able to locate these records directly
- Think about an example in a library:
  - How can we find a book?
    - Books are on the shelves in a sequential order
    - We had **drawers** where you could **look for books** by author, title or sometimes subject that were telling you the "**coordinates**" of a book
  - author, title and subject are like indexes (索引)



# Terminology

- Plural of index: indices, or indexes?
  - Both are correct in English
    - indices (Latin): Often used in scientific and mathematical context representing the places of an element in an array, vector, matrix, etc.
    - indexes (American English): Used in publishing for the books
  - What about database?
    - A good way: Follow the naming convention of the project or the DBMS

[Chapter 11. Indexes](#)

[Prev](#) [Up](#)

[Part II. The SQL Language](#)

[Home](#) [Next](#)

## Chapter 11. Indexes

# Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key

```
1 1,12 stulyev,ru,1971,161
2 2,Al-mummia test,eg,1969,102
3 3,"Ali Zaoua, prince de la rue",ma,2000,90
4 4,Apariencias,ar,2000,94
5 5,Ardh Satya,in,1983,130
6 6,Armaan,in,2003,159
7 7,Armaan,pk,1966,
8 8,Babette's gæstebud,dk,1987,102
9 9,Banshun,jp,1949,108
10 10,Bidaya wa Nihaya,eg,1960,
11 11,Variety,us,2008,106
12 12,"Bon Cop, Bad Cop",ca,2006,
13 13,Brilliantovaja ruka,ru,1969,100
14 14,C'est arrivé près de chez vous,be,1992,95
15 15,Carlota Joaquina - Princesa do Brasil,br,1995,
16 16,Cicak-man,my,2006,107
```

# Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key
  - E.g., find movies with IDs larger than 100 and smaller than 200

1	1,12 stulyev,ru,1971,161
2	2,Al-mummia test,eg,1969,102
3	3,"Ali Zaoua, prince de la rue",ma,2000,90
4	4,Apariencias,ar,2000,94
5	5,Ardh Satya,in,1983,130
6	6,Armaan,in,2003,159
7	7,Armaan,pk,1966,
8	8,Babette's gæstebud,dk,1987,102
9	9,Banshun,jp,1949,108
10	10,Bidaya wa Nihaya,eg,1960,
11	11,Variety,us,2008,106
12	12,"Bon Cop, Bad Cop",ca,2006,
13	13,Brilliantovaja ruka,ru,1969,100
14	14,C'est arrivé près de chez vous,be,1992,95
15	15,Carlota Joaquina - Princesa do Brasil,br,1995,
16	16,Cicak-man,my,2006,107

In the current storage structure, the records are sorted by movieid

- So, it will be easy to find a specific movieid with binary search

# Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key
- However, how can we find data based on the **non-sorted columns**?
  - E.g., find all Chinese movies

```
1 1,12 stulyev,ru,1971,161
2 2,Al-mummia test,eg,1969,102
3 3,"Ali Zaoua, prince de la rue",ma,2000,90
4 4,Apariencias,ar,2000,94
5 5,Ardh Satya,in,1983,130
6 6,Armaan,in,2003,159
7 7,Armaan,pk,1966,
8 8,Babette's gæstebud,dk,1987,102
9 9,Banshun,jp,1949,108
10 10,Bidaya wa Nihaya,eg,1960,
11 11,Variety,us,2008,106
12 12,"Bon Cop, Bad Cop",ca,2006,
13 13,Brilliantovaja ruka,ru,1969,100
14 14,C'est arrivé près de chez vous,be,1992,95
15 15,Carlota Joaquina - Princesa do Brasil,br,1995,
16 16,Cicak-man,my,2006,107
```

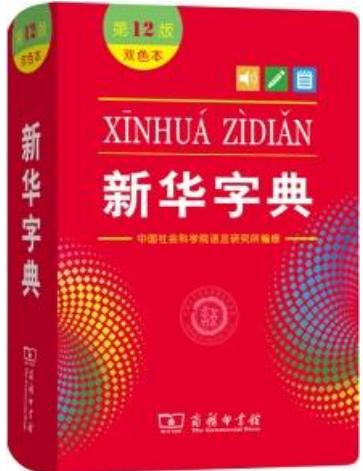
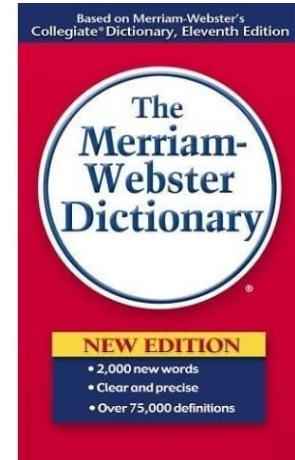
country

Find the rows where country = 'cn'

- The country codes are not sorted in the current storage structure, so the binary search algorithm cannot be used

# Searching for Record

- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are usually sorted in the alphabetical order of Pinyin
    - However, we have other ways of looking up a character
      - Radicals (偏旁部首)
      - Number of strokes (数笔画)
      - Four-corner method (四角号码)



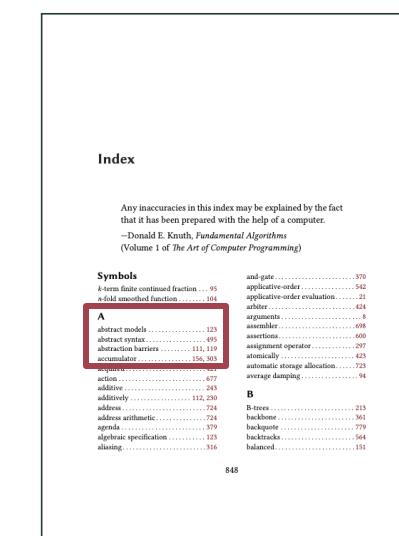
# Practical Use

# Index in Databases

- Concept
  - An **index** is a **data structure** which improves the efficiency of retrieving data with specific values from a database
  - Usually, indexes **locate a row** by a series of location indicators
    - E.g., (filename, block number, offset)

# Index in Databases

- Concept
  - An **index** is a **data structure** which improves the efficiency of retrieving data with specific values from a database
  - Usually, indexes locate a row by a series of location indicators
    - E.g., (filename, block number, offset)
- It is like indexes in books
  - Location indicator: (page, row)



**A**

abstract models ..... 123  
abstract syntax ..... 495  
abstraction barriers ..... 111, 119  
accumulator ..... 156, 303

**B**

Bitwise ..... 215  
backbone ..... 361  
backnote ..... 779  
buckets ..... 564  
balanced ..... 151

# Index in Databases

- Actually, we have been benefited from indexes off-the-shelf



```
▼ └── indexes 2
    └── movies_pkey (movieid) UNIQUE
    └── movies_title_country_year_released_key (title, country, year_released) UNIQUE
```

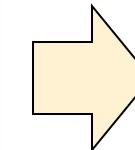
- In PostgreSQL, indexes are built automatically on columns with **primary key** or **unique** constraints

# Experiment on Using Indexes

- Duplicate a table with no index



```
create table movies_no_index as select * from movies;
```



```
-- auto-generated definition
create table movies_no_index
(
    movieid      integer,
    title        varchar(100),
    country      char(2),
    year_released integer,
    runtime      integer,
    user_name    varchar(20)
);
```

# Experiment on Using Indexes

- Check the performance on retrieving data
  - Significant difference between queries on the two tables

```
-- Query 1  
explain analyze  
select *  
from movies  
where movieid > 100 and movieid < 300;  
  
-- Query 2  
explain analyze  
select *  
from movies_no_index  
where movieid > 100 and movieid < 300;
```

Query 1  
(on `movies`)

```
QUERY PLAN  
1 Bitmap Heap Scan on movies  (cost=10.32..136.35 rows=199 width=40) (actual time=0.162..0.440 rows=199 loops=1)  
2  Recheck Cond: ((movieid > 100) AND (movieid < 300))  
3  Heap Blocks: exact=6  
4  -> Bitmap Index Scan on movies_pkey  (cost=0.00..10.28 rows=199 width=0) (actual time=0.136..0.136 rows=199 loops=1)  
5    Index Cond: ((movieid > 100) AND (movieid < 300))  
6 Planning Time: 0.413 ms  
7 Execution Time: 0.507 ms
```

Query 2  
(on `movies_no_index`)

```
QUERY PLAN  
1 Seq Scan on movies_no_index  (cost=0.00..217.06 rows=199 width=40) (actual time=0.039..5.075 rows=199 loops=1)  
2  Filter: ((movieid > 100) AND (movieid < 300))  
3  Rows Removed by Filter: 9005  
4 Planning Time: 0.444 ms  
5 Execution Time: 5.156 ms
```

# Experiment on Using Indexes

- If there is no index on a column (or several columns), we can create one manually



```
-- SQL Syntax for creating indexes
create index index_name
on table_name (column_name [, ...]);
```

# Theoretical Aspects

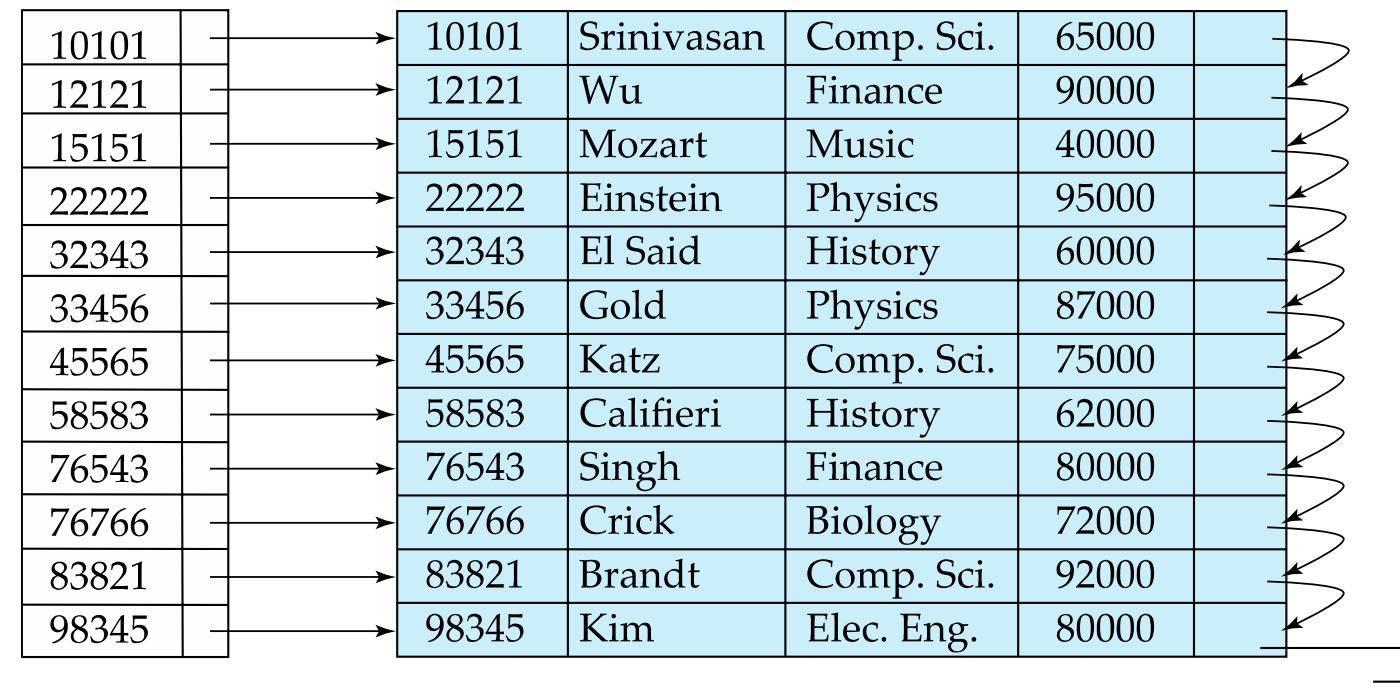
# Index Taxonomy

- 1) In terms of storage structure, is the index completely separated with the data records?
  - No ⇒ **Integrated index**
    - PK (primary key) index in a MySQL InnoDB database
    - PK index in a SQL Server database
  - Yes ⇒ **External index**
    - Indexes in a PostgreSQL database
    - Indexes in a MySQL MyISAM database

# Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

Example of  
clustered index

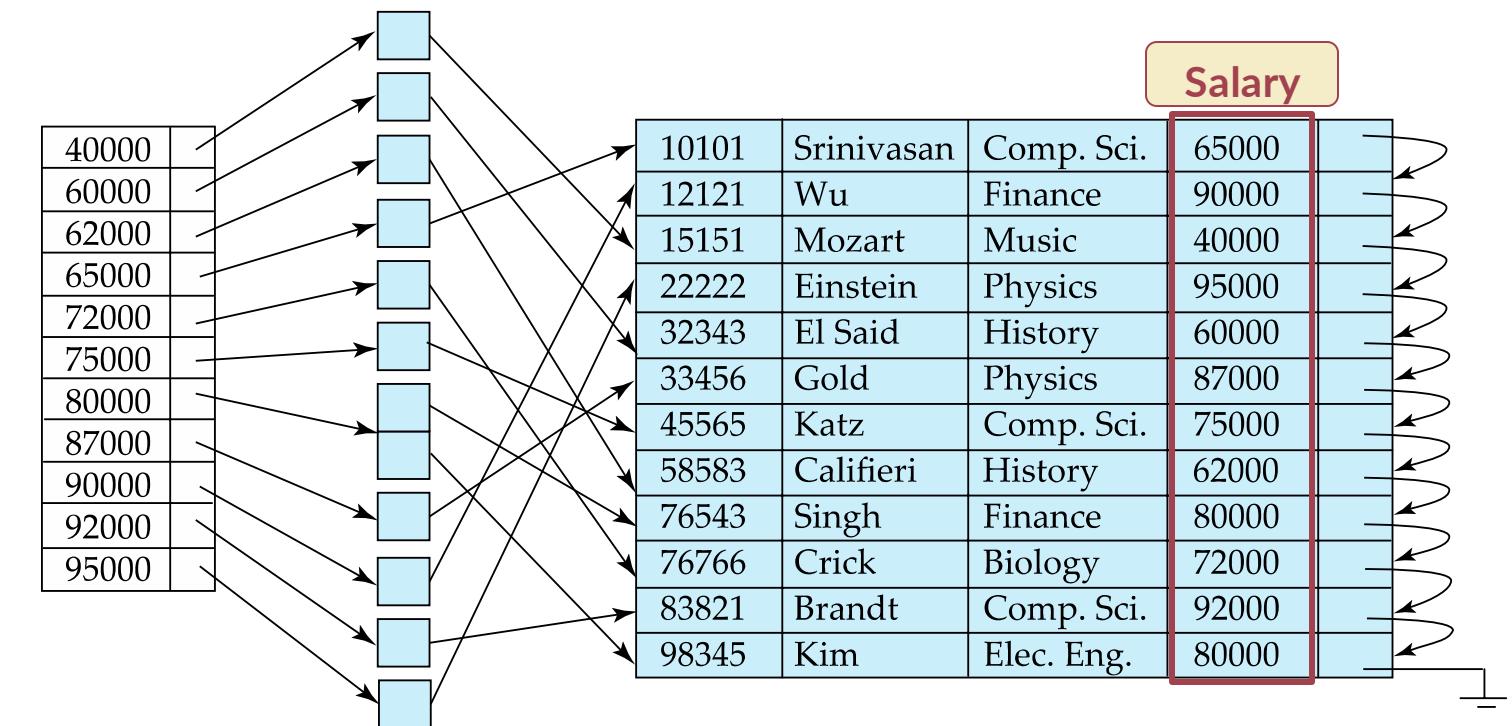


# Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

A secondary index on the column “salary”

- Index record points to a **bucket** that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

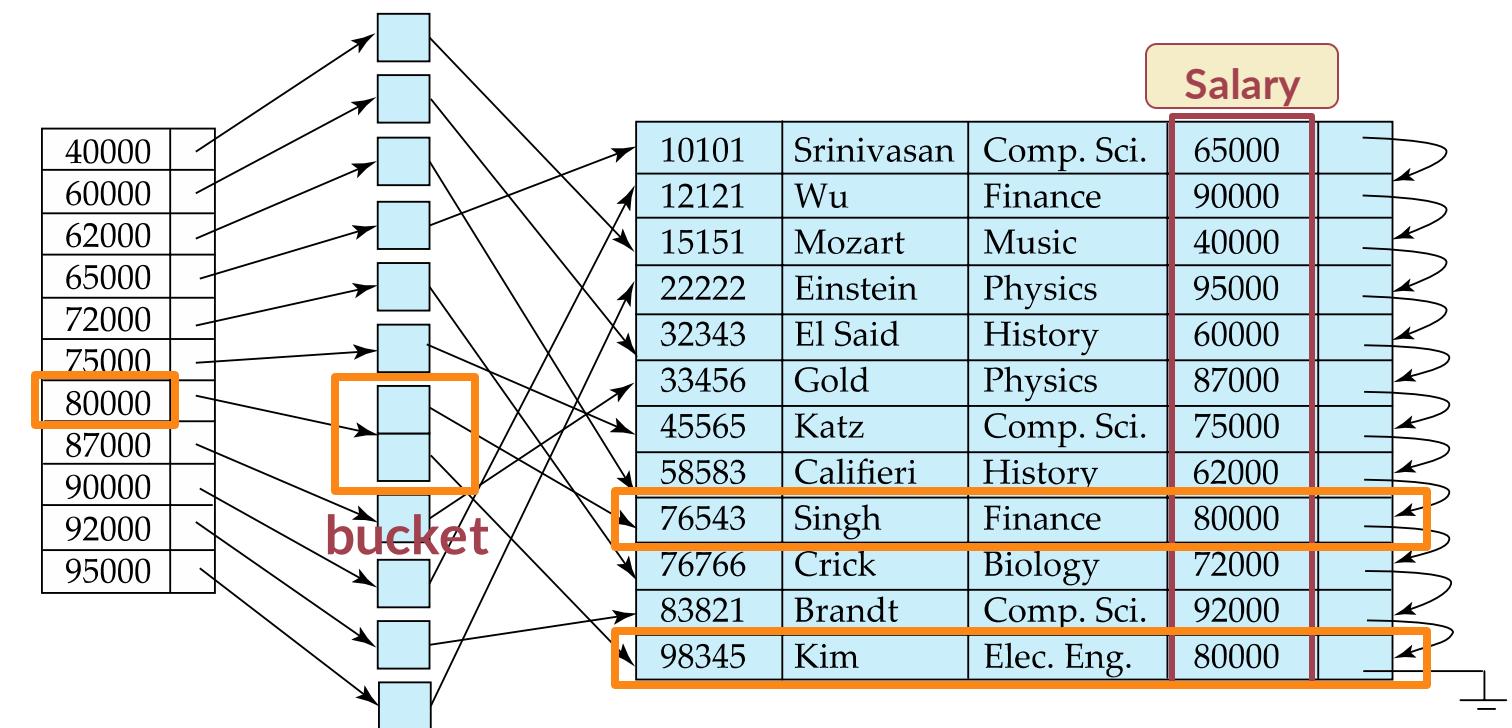


# Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
    - Allows the records of a file to be read in an order corresponding to the physical order in the file
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

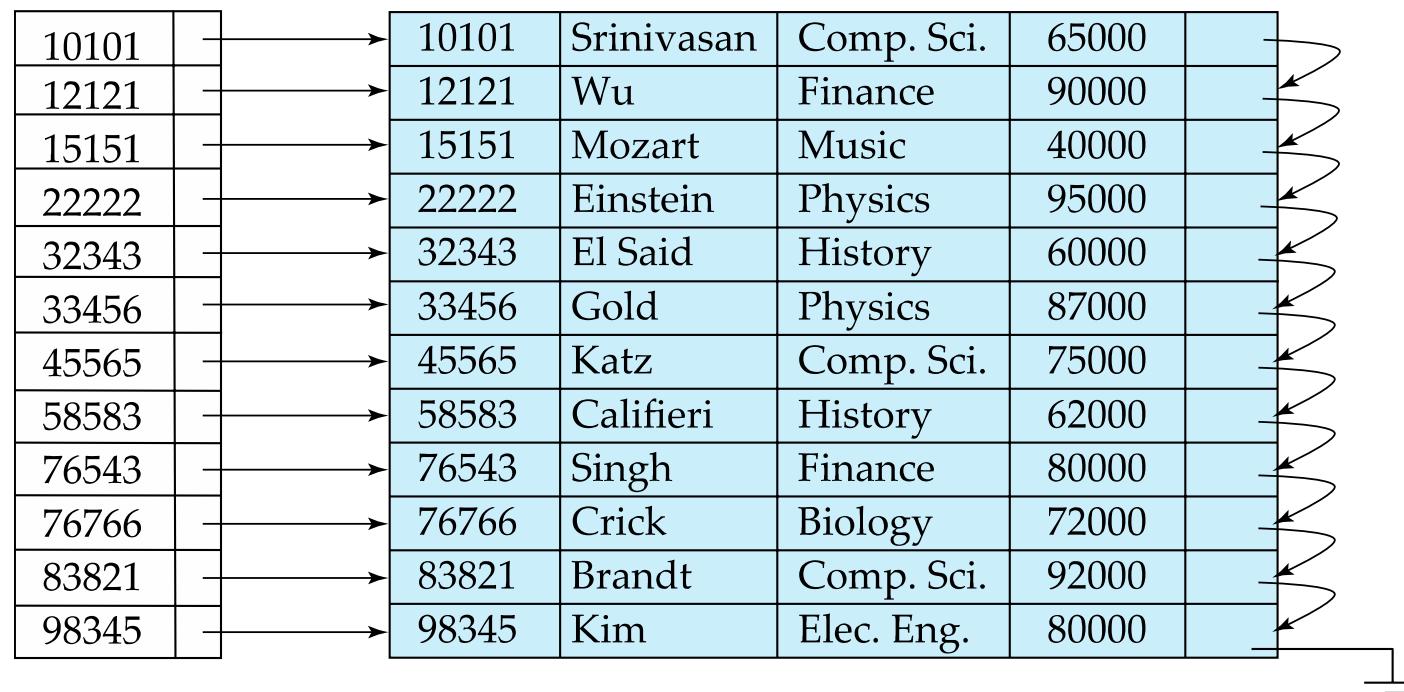
A secondary index on the column “salary”

- Index record points to a **bucket** that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

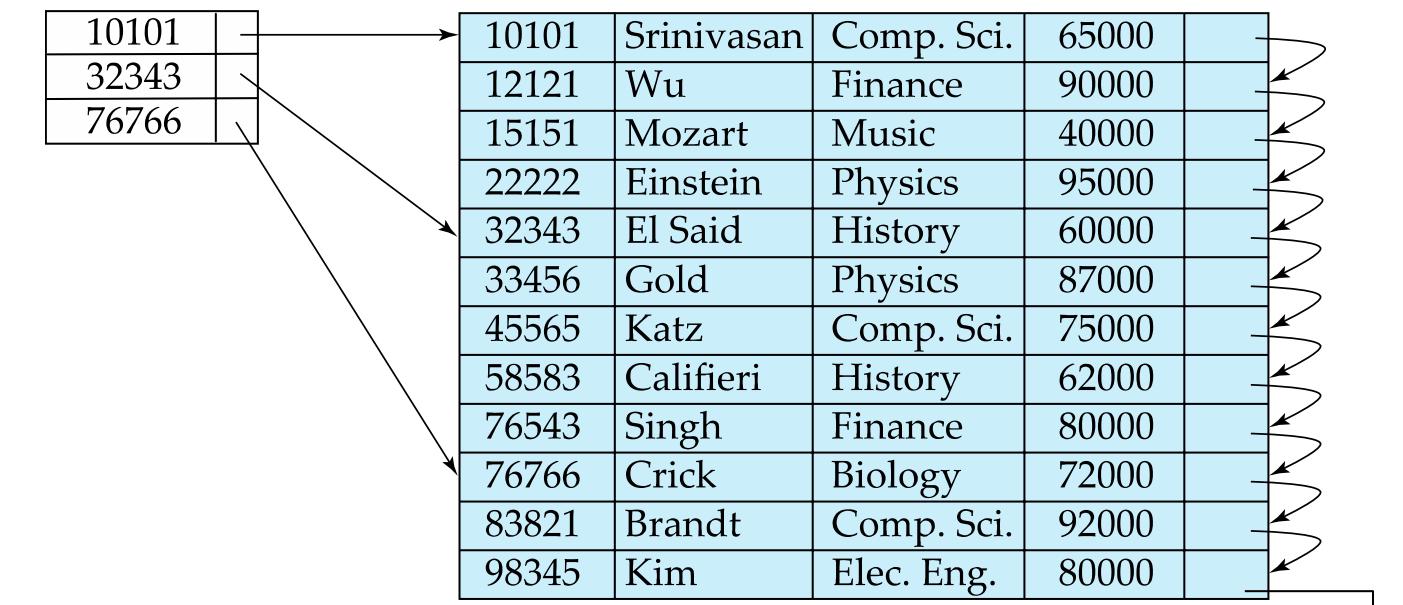


# Index Taxonomy

- 3) Does every search key in the data file correspond to an index entry?
  - Yes ⇒ **Dense Index**
  - No ⇒ **Sparse Index**



Dense Index



Sparse Index

# Index Taxonomy

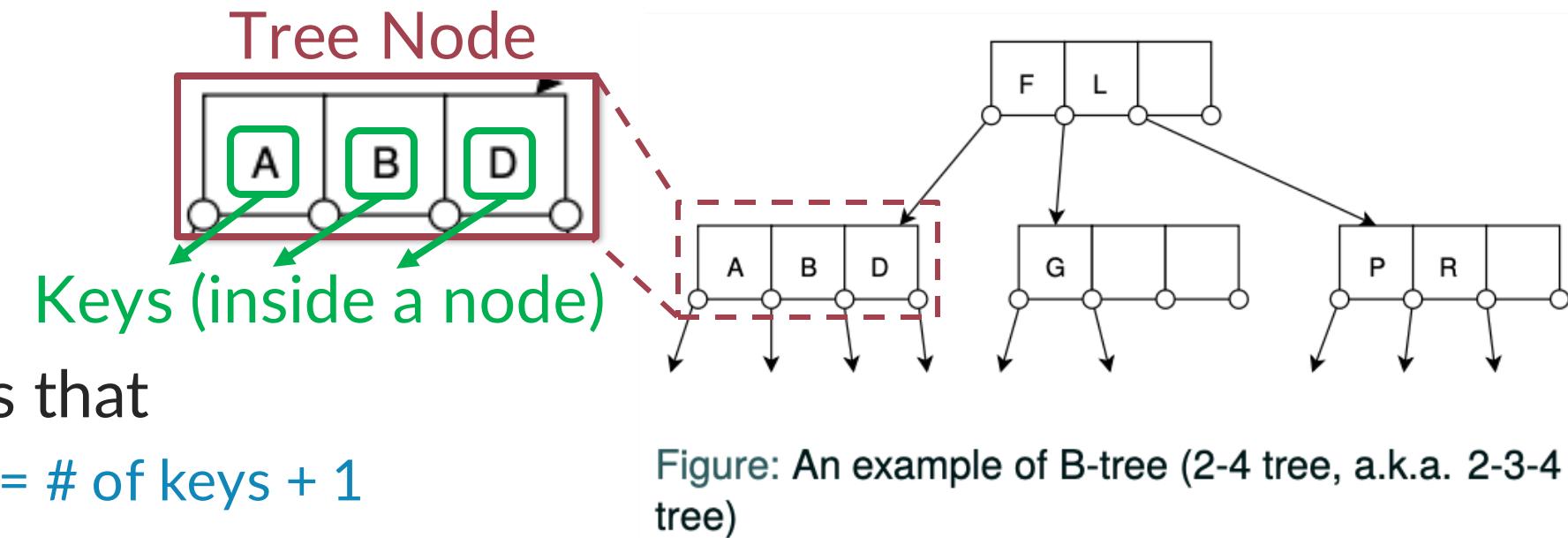
- 4) Does the search key contain more than one attribute?
  - Yes ⇒ **Multi-key index** (Multi-column index)
  - No ⇒ **Single-key index** (Single-column index)
    - *We mainly focus on single-key index for now*

# Index Implementation

- Data Structures for Indexes
  - B-tree, B+-tree
    - Very famous data structures for building indexes
  - Hash table

# B-tree

- A B-tree of **order  $m$**  satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing  $n$  keys ( $K_1 < K_2 < K_3 < \dots < K_n$ ) with  $n+1$  children (pointed by  $P_0, P_1, P_2, \dots, P_n$ ), any key  $k_{\text{sub } i}$  in the sub-tree pointed by  $P_i$  satisfies that  $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**Multiway**) For an internal node,  $\lceil m/2 \rceil \leq \# \text{ of children} \leq m$ 
    - ... except that a root node may have less than  $\lceil m/2 \rceil$  children
  - (**Always balanced**) All leaves appear on the same level



# B-tree

- A B-tree of **order  $m$**  satisfies that
  - For every node, **# of children = # of keys + 1**
  - (**Ordered**) For a node containing  $n$  keys ( $K_1 < K_2 < K_3 < \dots < K_n$ ) with  $n+1$  children (pointed by  $P_0, P_1, P_2, \dots, P_n$ ), any key  $k_{\text{sub } i}$  in the sub-tree pointed by  $P_i$  satisfies that  $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**Multiway**) For an internal node,  $\lceil m/2 \rceil \leq \# \text{ of children} \leq m$ 
    - ... except that a root node may have less than  $\lceil m/2 \rceil$  children
  - (**Always balanced**) All leaves appear on the same level

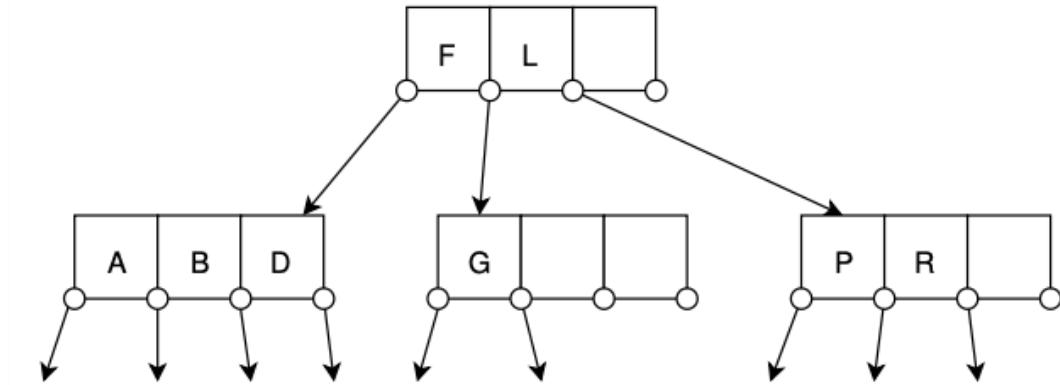


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

# B-tree

- A B-tree of **order  $m$**  satisfies that
  - For every node, **# of children = # of keys + 1**
  - (**Ordered**) For a node containing  $n$  keys ( $K_1 < K_2 < K_3 < \dots < K_n$ ) with  $n+1$  children (pointed by  $P_0, P_1, P_2, \dots, P_n$ ), any key  $k_{\text{sub } i}$  in the sub-tree pointed by  $P_i$  satisfies that  $K_i < k_{\text{sub } i} < K_{i+1}$
  - (**Multiway**) For an internal node,  $\lceil m/2 \rceil \leq \# \text{ of children} \leq m$ 
    - ... except that a root node may have less than  $\lceil m/2 \rceil$  children
  - (**Always balanced**) All leaves appear on the same level

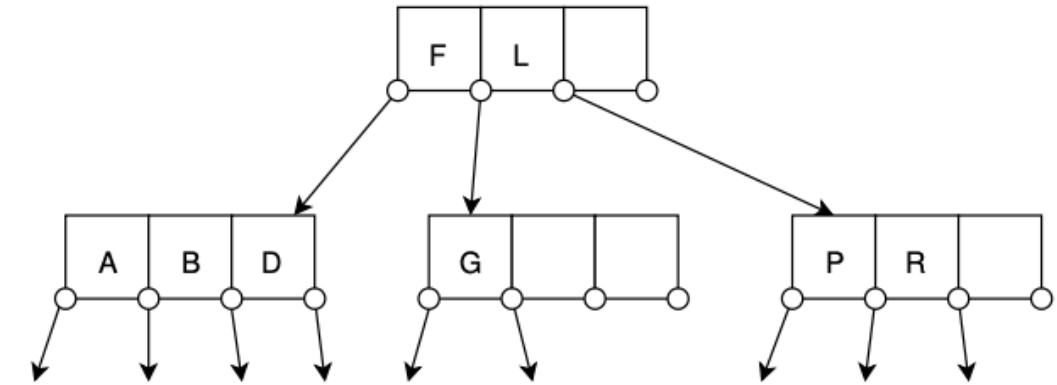


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

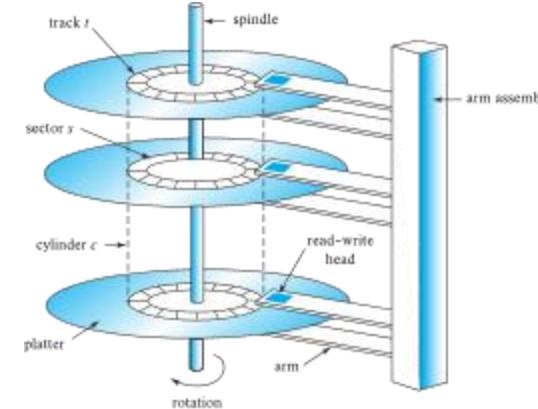
- $\lceil m/2 \rceil$  is called the **minimum branching factor** (a.k.a. **minimum degree**) of the tree
- A B-tree of order  $m$  is usually called a " $\lceil m/2 \rceil$ - $m$  tree", like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, ...
  - In practice, the order  $m$  is much larger (~100)

# B-tree

- Height of a  $B$ -tree:  $h \leq 1 + \log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right)$
- If we take an 50-100 tree with 1M records:
  - $h \leq 1 + \log_{100/2}(1000000/2) = 4.354$  (i.e., 4 levels)

# B-tree

- Height of a *B*-tree:  $h \leq 1 + \log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right)$
- If we take an 50-100 tree with 1M records:
  - $h \leq 1 + \log_{100/2}(1000000/2) = 4.354$  (i.e., 4 levels)
- Why do we use B-trees?
  - We can set the size of a B-tree node as the disk page size
    - i.e.,  $m$  can be chosen with consideration on the page size
  - The height of the tree -> Number of disk I/Os
    - The number of disk I/Os can be relatively small



Access time: 5-20ms

1ns =  $10^{-6}$ ms



Access time: 50-70ns

Seconds:

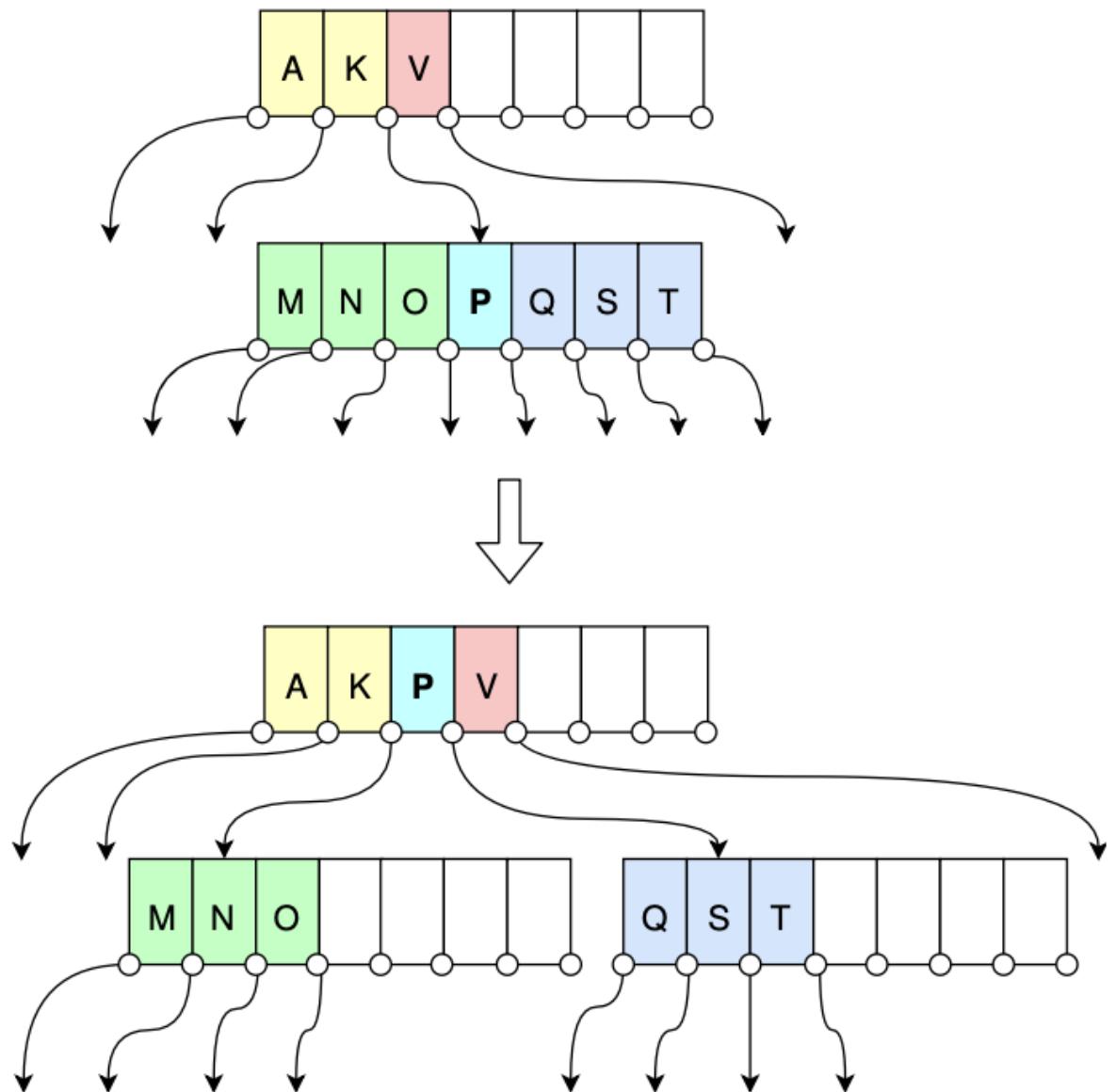
Hours:

# B-tree

- Tree operations:
  - Search, Insert, Delete
  - Update (Delete + Insert)
- What is special in B-tree
  - Split and merge nodes

# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$
  - ... and we want to insert the record with **key="P"**

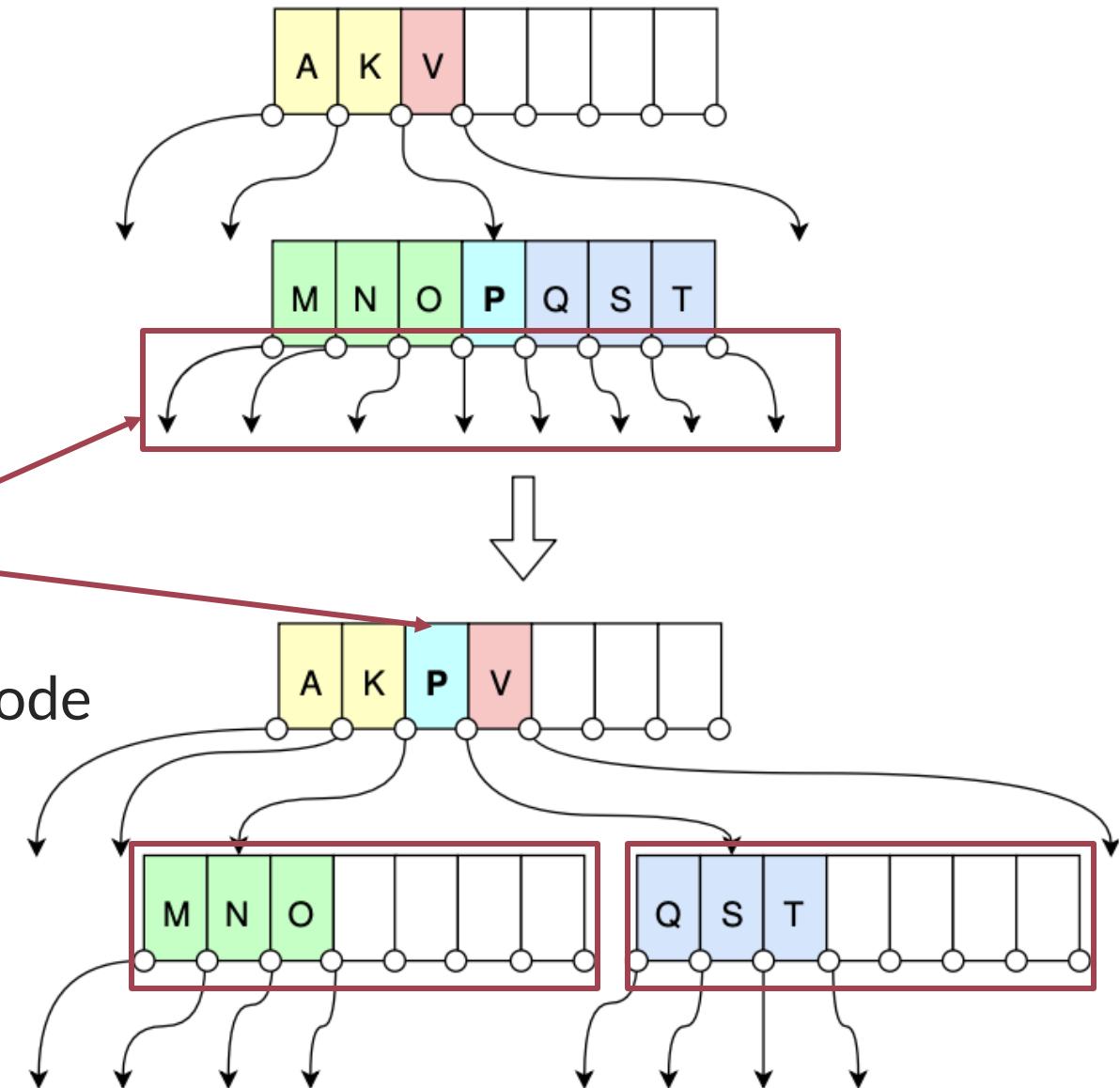


# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$
  - ... and we want to insert the record with **key="P"**

The number of children is larger than  $m$  ( $m=7$ )

- This node will be split into two nodes
- The pivot key will be elevated into the parent node



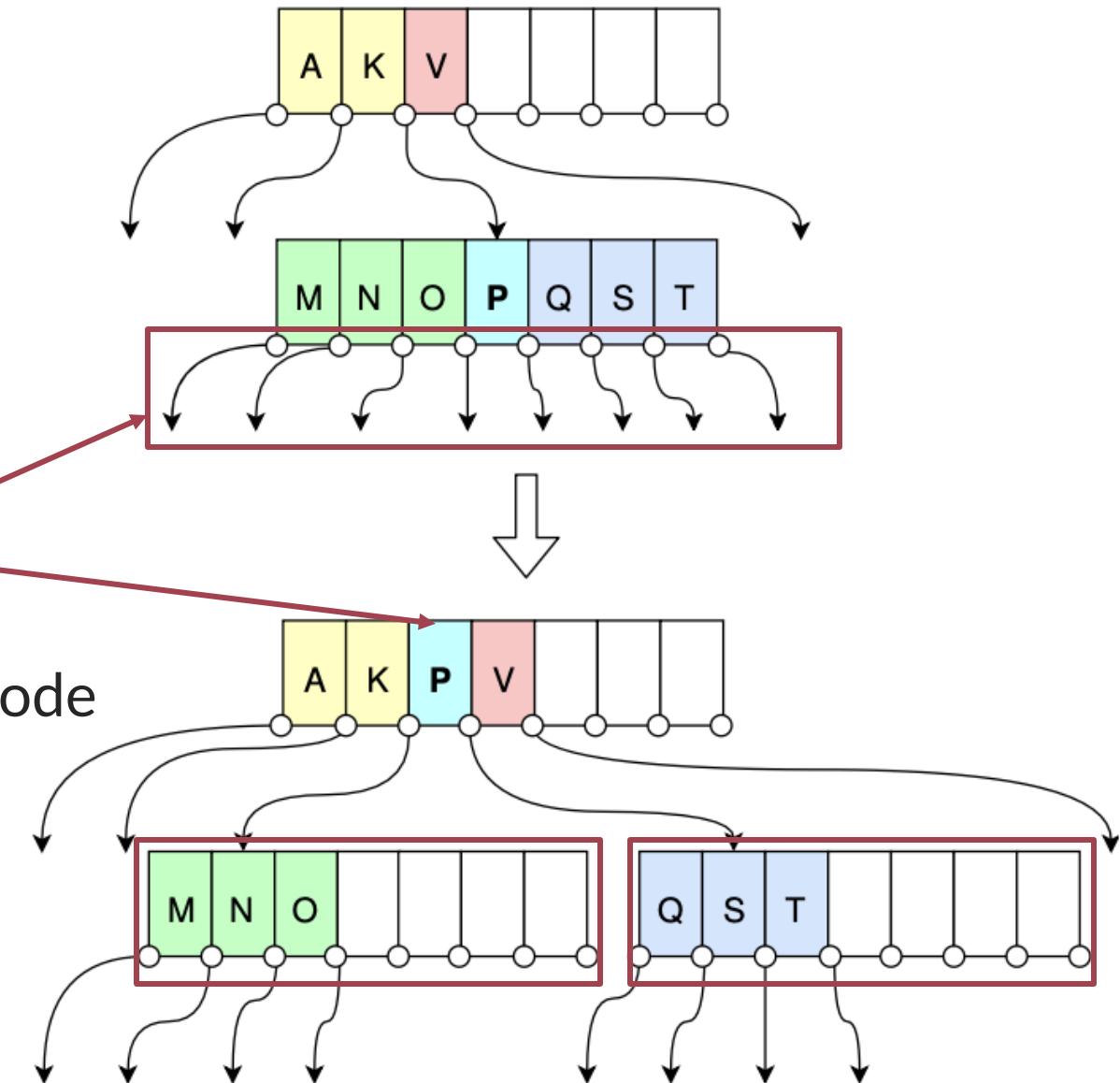
# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$
  - ... and we want to insert the record with **key="P"**

The number of children is larger than  $m$  ( $m=7$ )

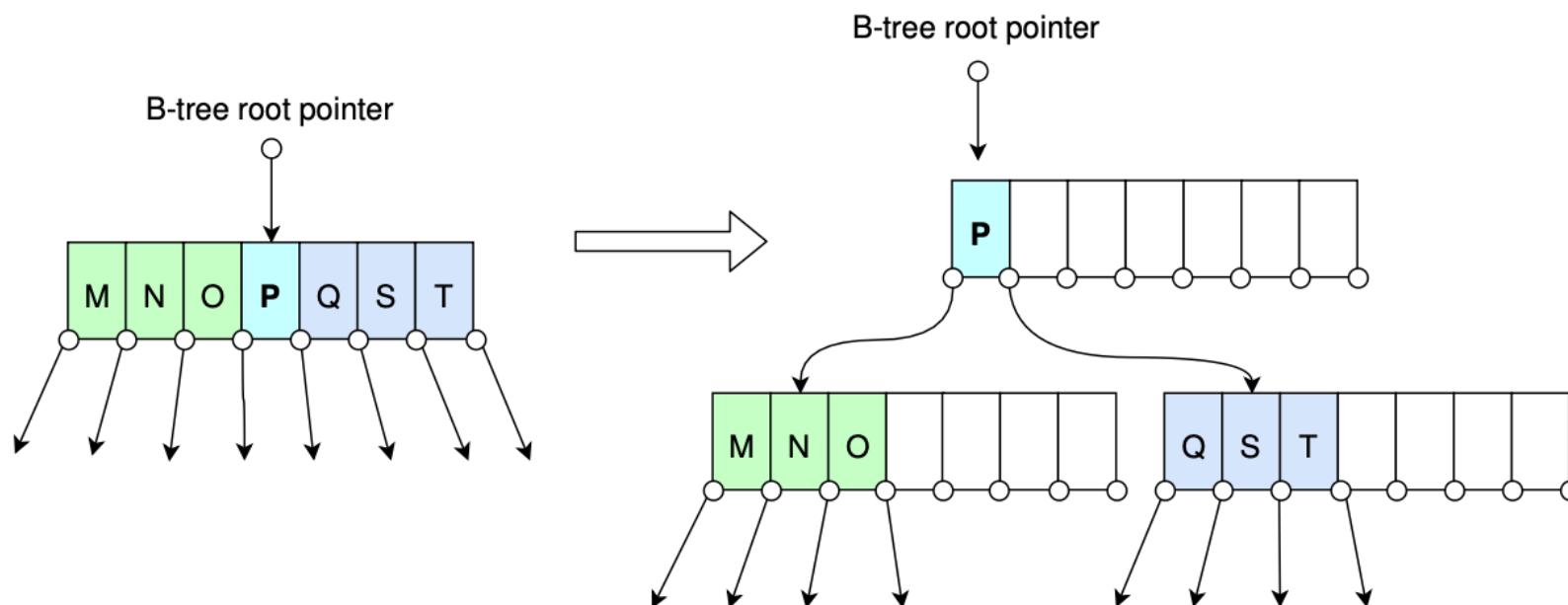
- This node will be split into two nodes
- The pivot key will be elevated into the parent node

- What if the parent (or even the root) node is also full?



# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$
  - ... and we want to insert the record with key="P"
- Split the root node of the B-tree



Note that the height of the B-tree is increased by 1 in this case

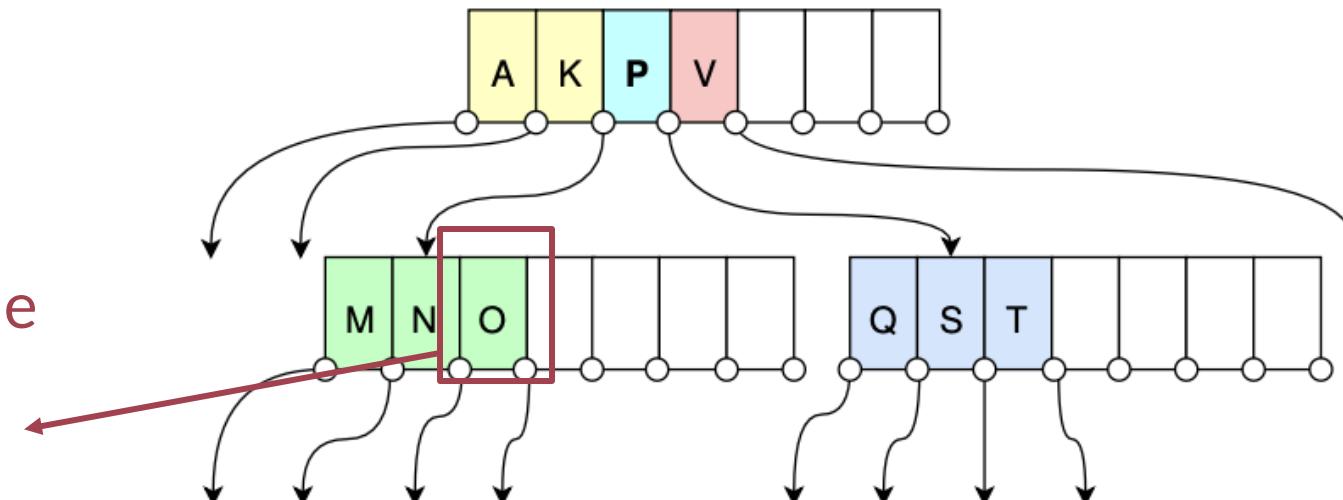
- This is the only way that a B-tree increases its height

# B+-tree

- A Problem in B-tree: **Table traversal** when only the B-tree is provided
  - In B-tree, data are stored on all nodes
  - What if we want to traverse all records in the table?
    - `select * from letters`

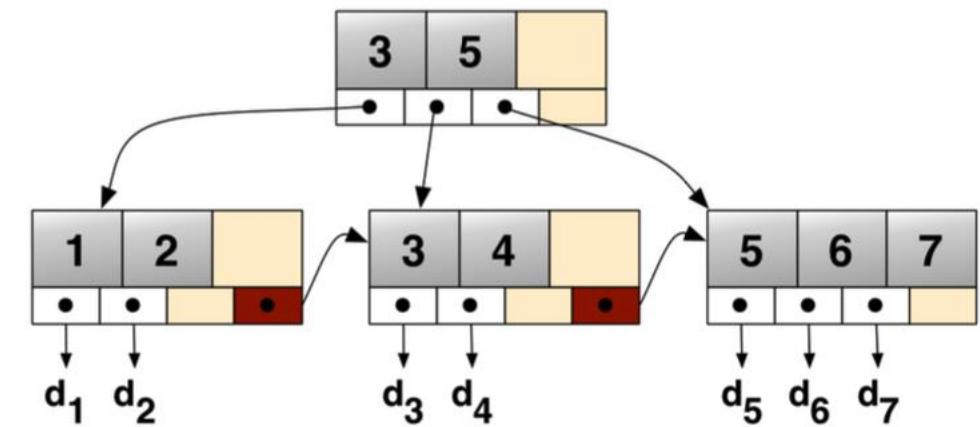
For example, we have accessed the node for letter "O"

- How can we find the next row?
  - We must go back to the parent node to access "P" (extra time cost)



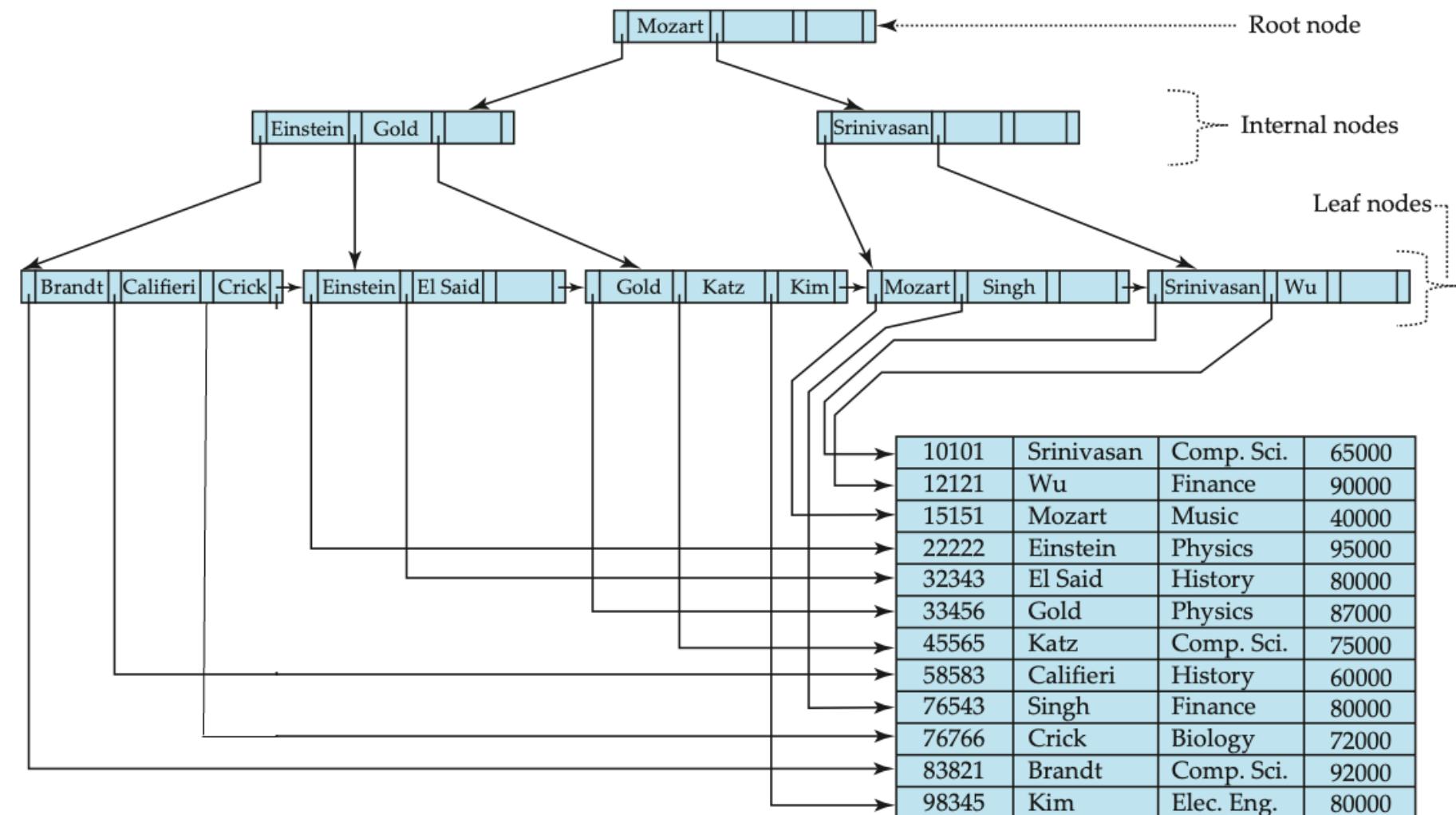
# B+-tree

- Features of a B+-tree (compared with B-trees)
  - Data stored **only in leaves**
  - Leaves are linked sequentially



# B+-tree

- A complete example of a B+-tree
  - Data stored **only in leaves**
    - No need to squeeze data into non-leaf nodes
  - Leaves are linked sequentially
    - **Faster table traversal** from top to bottom
    - Better support for range queries



# Index It or Not: Where Indexing May Help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria

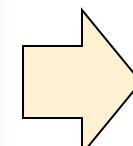
- =
- <, <=, >, >=, between
- in

- exists
- like (prefix matching)

- Be cautious when the indexed columns need frequent writing operations
  - Overhead to update indexes in `insert`, `update`, and `delete` operations
- Functions



```
SELECT attr1, attr2  
FROM table  
WHERE function(column) = search_key
```



```
-- Create an index on the return values of the function  
-- instead of the original values  
create index idx_name ON table1(function(col1));
```

Note: The expression should be deterministic. For detailed usage, please refer to:  
<https://www.postgresql.org/docs/14/indexes-expressional.html>

# Index It or Not: Where Indexing May Help

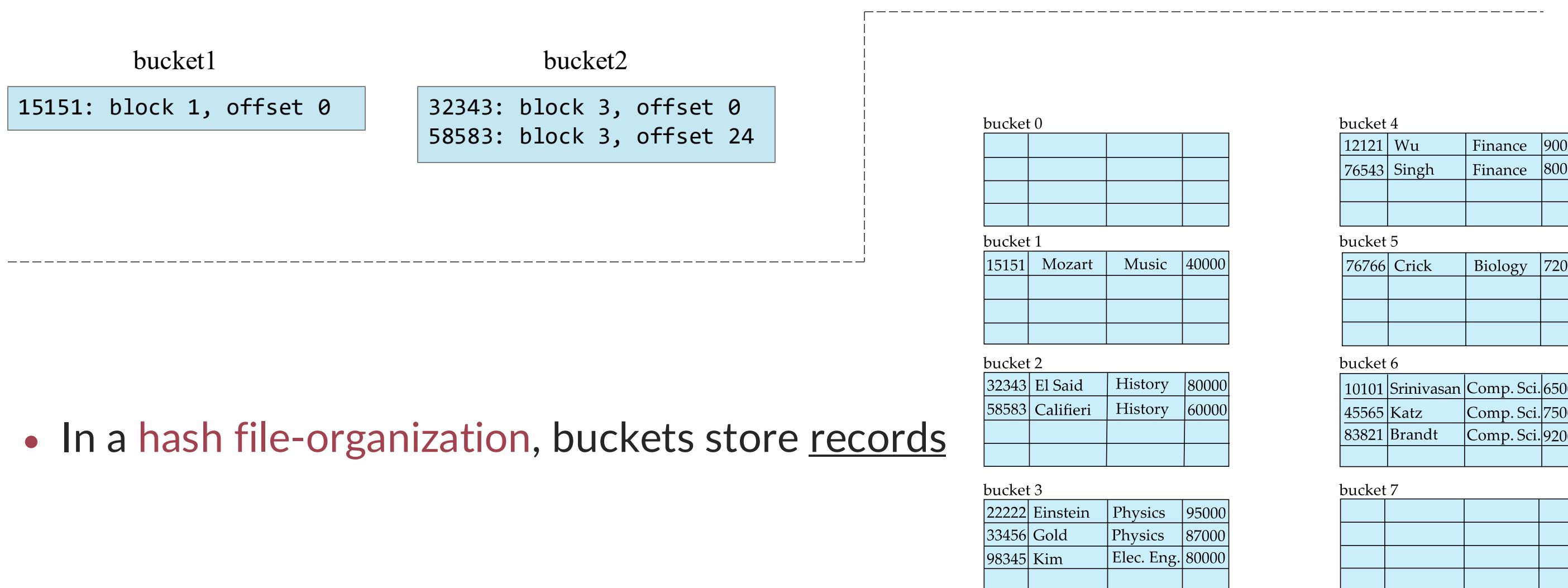
- Be cautious when using indexes on a small table
  - Full scan ≠ Bad scheme
  - Index retrieval ≠ Good scheme

# Hashing

- Hashing is a widely used technique for building indexes
- A **bucket** is a unit of storage containing one or more entries
  - A bucket is typically a disk block
  - We obtain the bucket of an entry from its search-key value using a **hash function**
    - Hash function  $h$  is a function from the set of all search-key values K to the set of all bucket addresses B
    - Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket
  - ... thus, the entire bucket must be searched sequentially to locate an entry.

# Hashing Index & Hashing File Organization

- In a **hash index**, buckets store entries with pointers to records



# Example: How join Works (with the help of indexes)

- Some widely used join algorithms
  - Nested-loop join
  - Hash join
  - Sort-merge join

# Example: How join Works (with the help of indexes)

- Nested (loop) join
  - Straight-forward linking between records from two tables in a nested-loop manner



```
for each row in t1 match C1(t1)
    for each row in t2 match P(t1, t2)
        if C2(t2)
            add t1|t2 to the result
```

# Example: How join Works (with the help of indexes)

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup
- Procedure:
  - 1. Create a hash table for the smaller table **t1** in the memory
  - 2. Scan the larger table **t2**. For each record **r**,
    - 2.1 Compute the hash value of **r.join\_attribute**
    - 2.2 Map to corresponding rows in **t1** using the hash table

# Example: How join Works (with the help of indexes)

- Sort-merge join (a.k.a. merge join)
  - Zipper-like joining
- Procedure:
  - 1. Sort tables  $t_1$  and  $t_2$  respectively according to the join attributes
  - 2. Perform an interleaved scan of  $t_1$  and  $t_2$ . When encountering a matched value, join the related rows together.

	$a1$	$a2$
$r$	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6

	$a1$	$a3$
$s$	a	A
	b	G
	c	L
	d	N
	m	B

When there are clustered indexes on the join attributes, step 1, the most expensive operation, can be skipped because  $t_1$  and  $t_2$  are already sorted in this scenario.