

# **Principles of Database Systems (CS307)**

## **Lecture 14: Indexing**

**Ran Cheng**

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

- Most contents are from slides made by Stéphane Faroult, Dr Yuxin Ma 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.

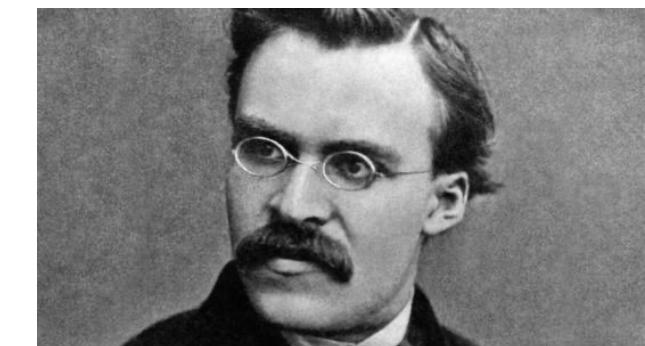
# 关于阿根廷



- 2014年，一觉醒来，梅西输了
- 昨天，看到最后，热泪盈眶

埃米利亚诺·马丁内斯：18岁加入阿森纳，但此后一直被外租，长期踢英冠；28岁，真正踢上英超联赛；29岁，首次入选阿根廷国家队，捧起美洲杯冠军；30岁，代表阿根廷出征世界杯，决赛最后时刻扑出法国准绝杀，点球大战中又扑出一个关键点球，捧起世界杯冠军。

“谁终将声震人间，必长久深自缄默；谁终将点燃闪电，必长久如云漂泊。”  
--弗里德里希·威廉·尼采



# Indexing

# Motivation

- 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 **what were the "coordinates" of a book.**



# 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
```

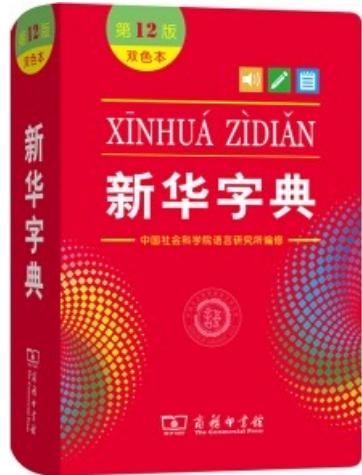
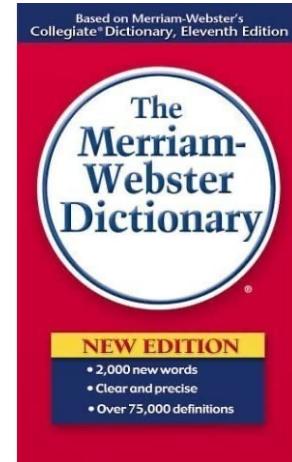
country

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

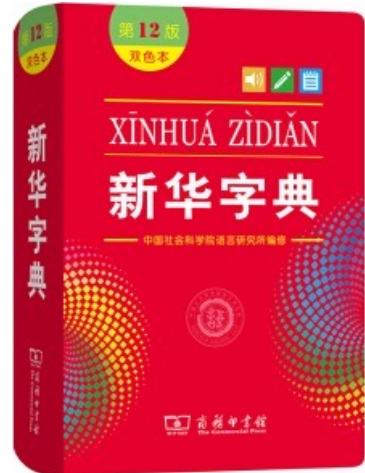
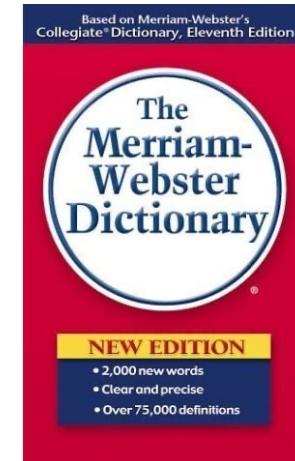
# Searching for Record

- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are sorted in the alphabetical order of Pinyin



# Searching for Record

- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are 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)

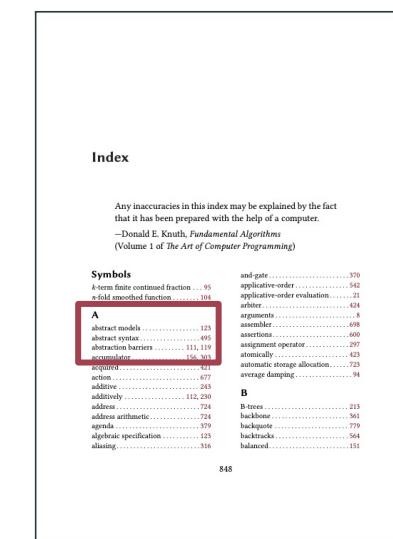


# Index in Databases

- Concept
  - An **index** is a **data structure** which improves the efficiency of retrieving data with specific values from a database
  - Usually, indices 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 indices in books
  - Location indicator: (page, row)



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

# Index in Databases

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



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

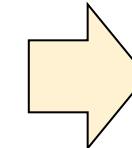
- In PostgreSQL, indexes are built automatically on columns with **primary key** or **unique** constraints

# 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 [, ...]);
```

# Index Taxonomy

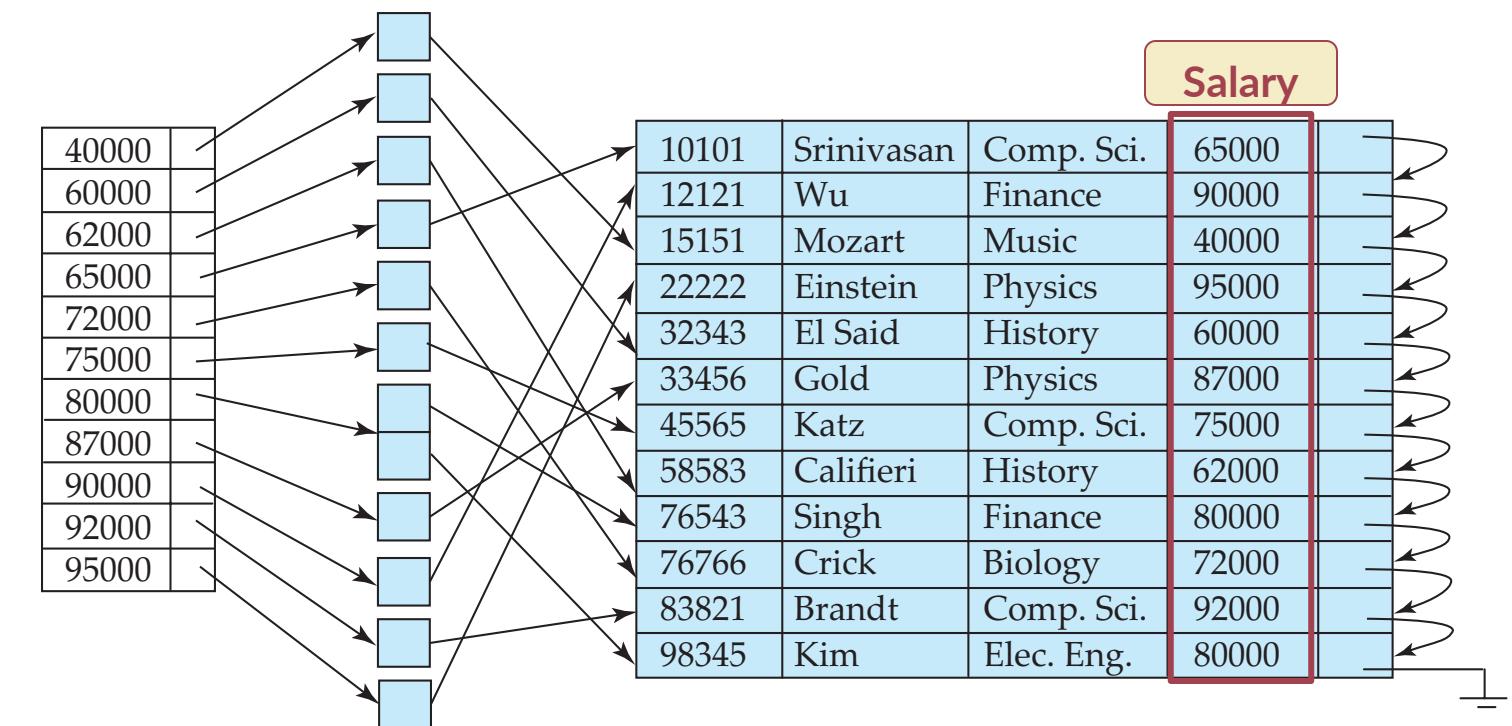
- 1) In terms of storage structure, is the index completely separated with the data records?
  - No ⇒ **Integrated index**
    - Primary Key Indices in a MySQL InnoDB database
    - Primary Key Indices in a SQL Server database
  - Yes ⇒ **External index**
    - Indices in a **PostgreSQL** database
    - Indices 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)
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

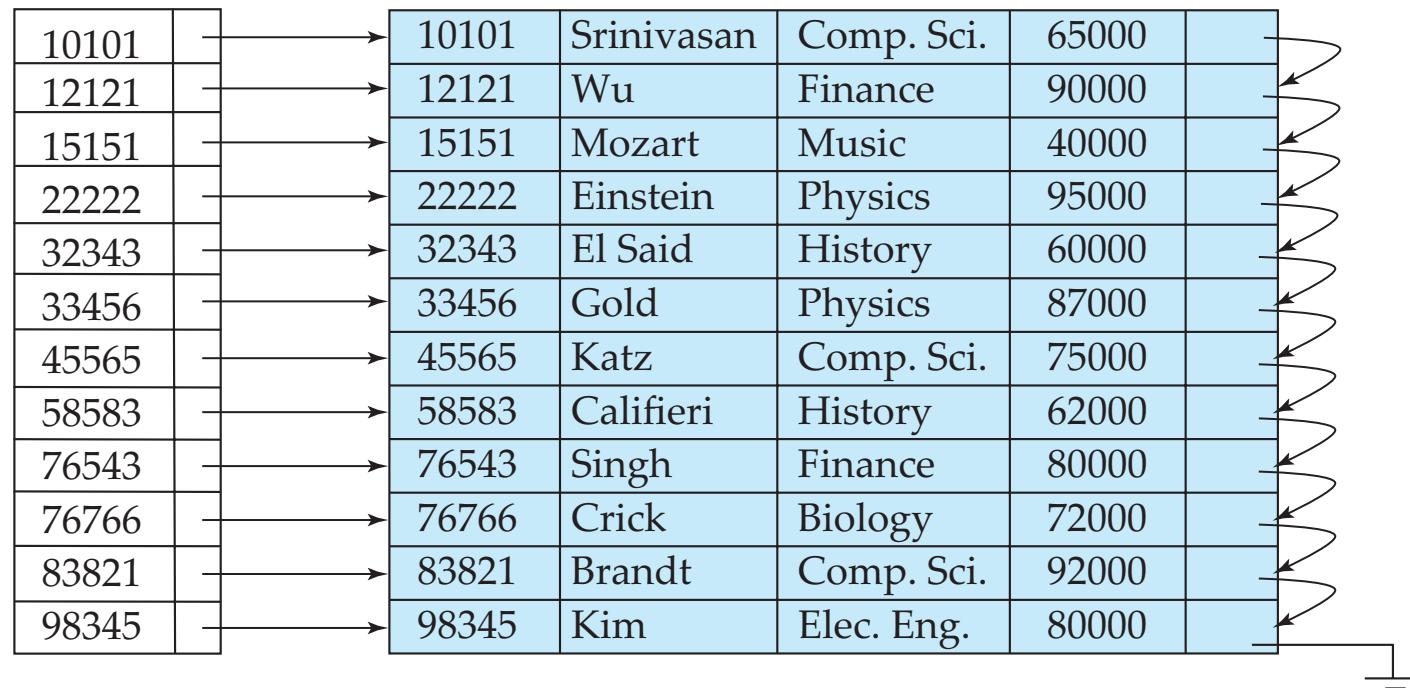
A secondary index on the column “salary”

- 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**

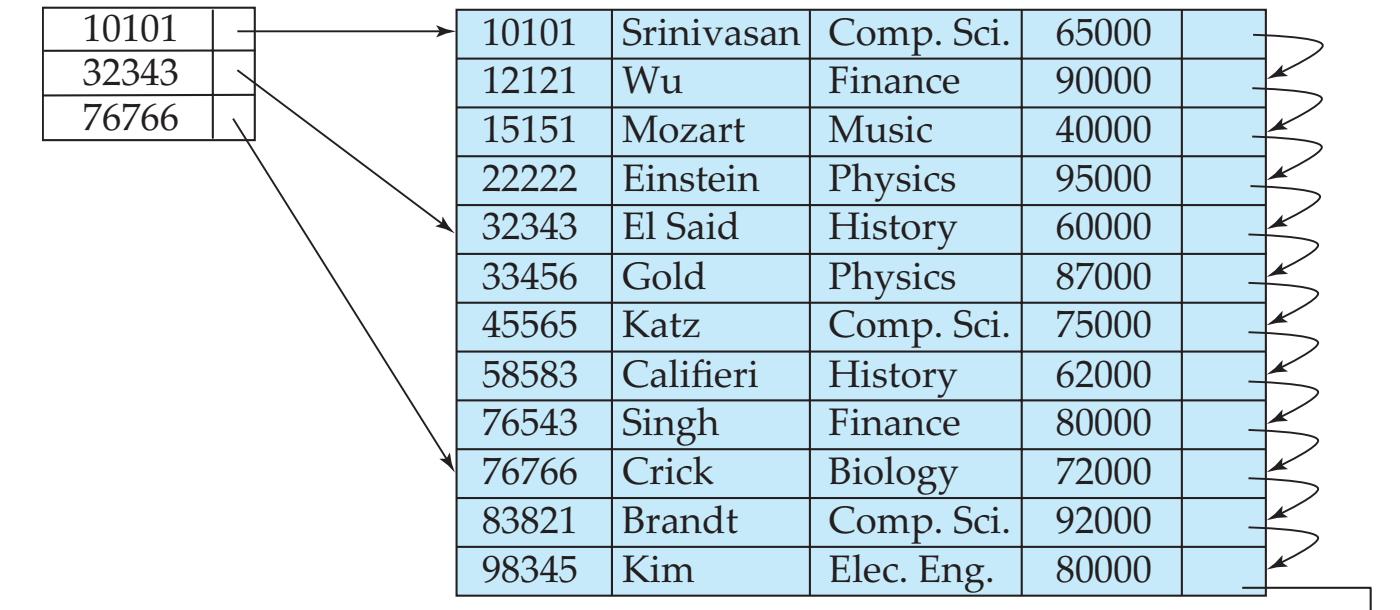


# Index Taxonomy

- 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 fewer 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

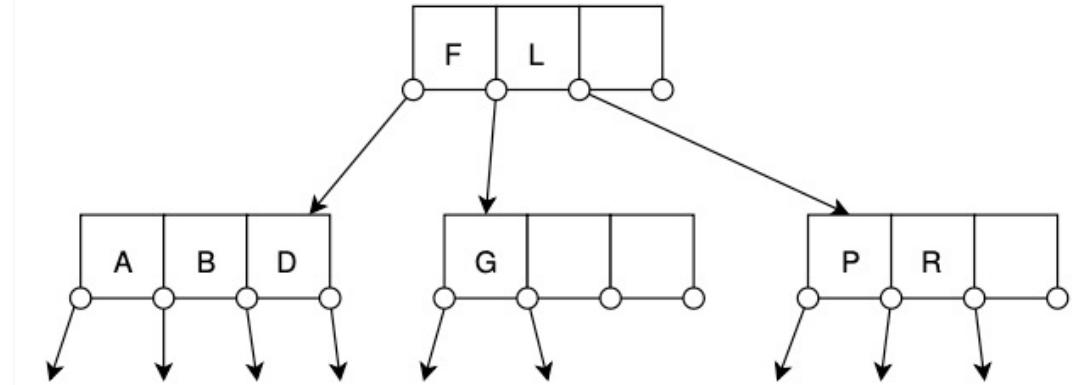


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

# B-tree

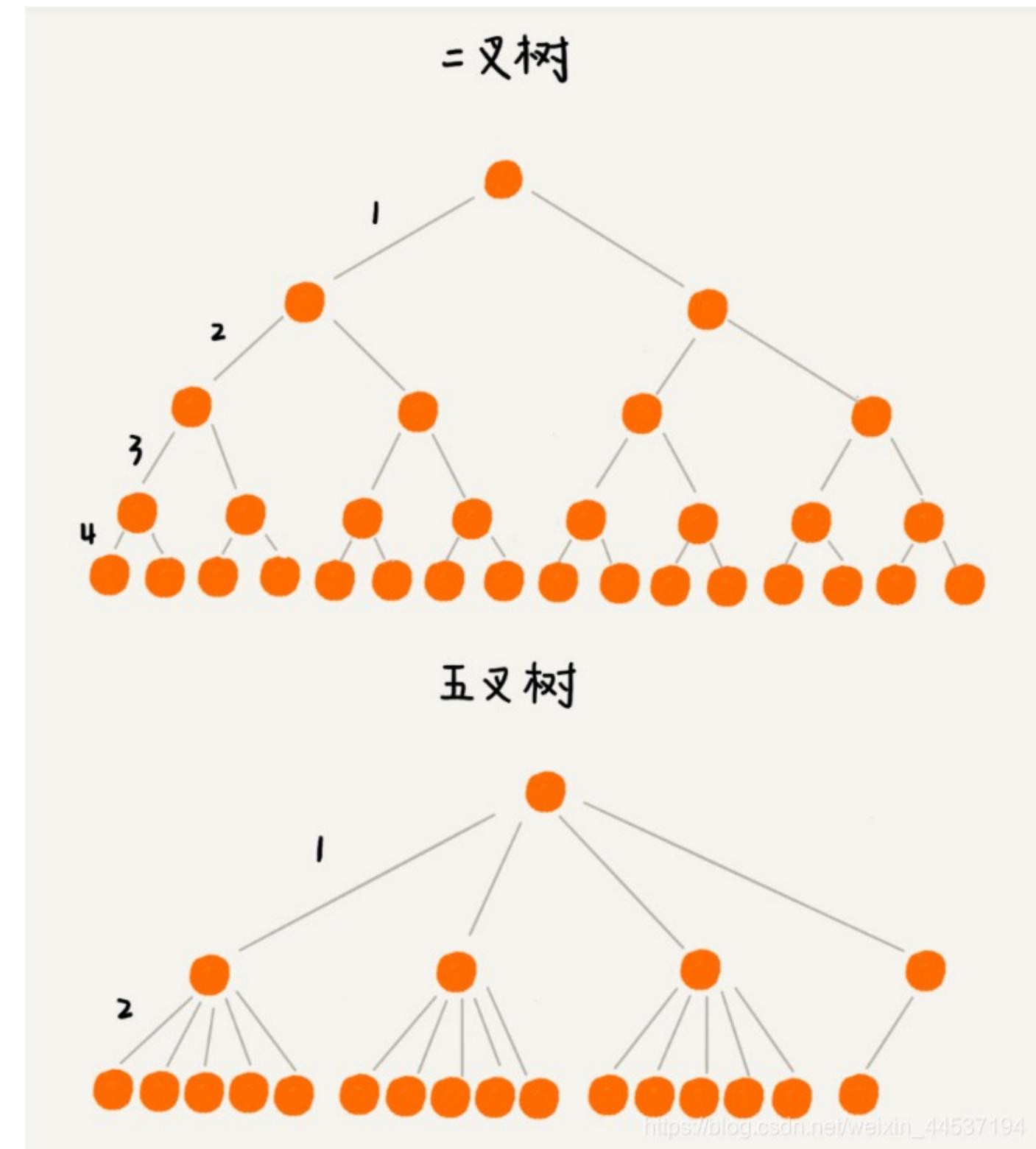
- A B-tree of order  $m$  satisfies that
    - For every node, # of children = # of keys + 1
    - (**Ordered**) For a node containing  $n$  keys ( $K_1 < K_2 < K_3 < \dots < K_n$ ) with  $n+1$  children (pointed by  $P_0, P_1, P_2, \dots, P_n$ ), any key  $k_{\text{sub } i}$  in the sub-tree pointed 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
- ```
graph TD; F["F | L"] --> G["G"]; F --> P["P | R"]; G --> H["H"]; G --> I["I"]; P --> Q["Q"]; P --> R["R"]; Q --> S["S"]; Q --> T["T"];
```

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

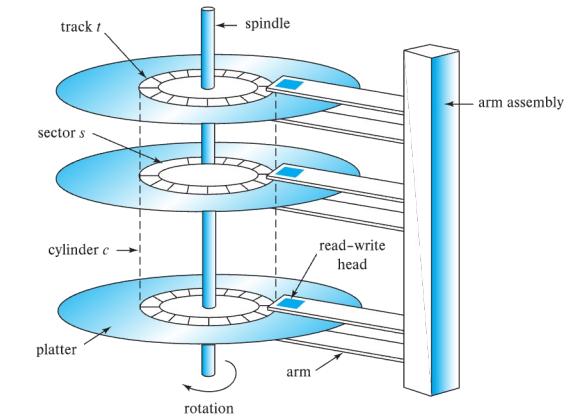
- Height of a  $m$ -order  $B$ -tree with  $n$  keywords?

$$\log_m n + 1 \leq h \leq \log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right) + 1$$



# 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



Seconds:

Hours:

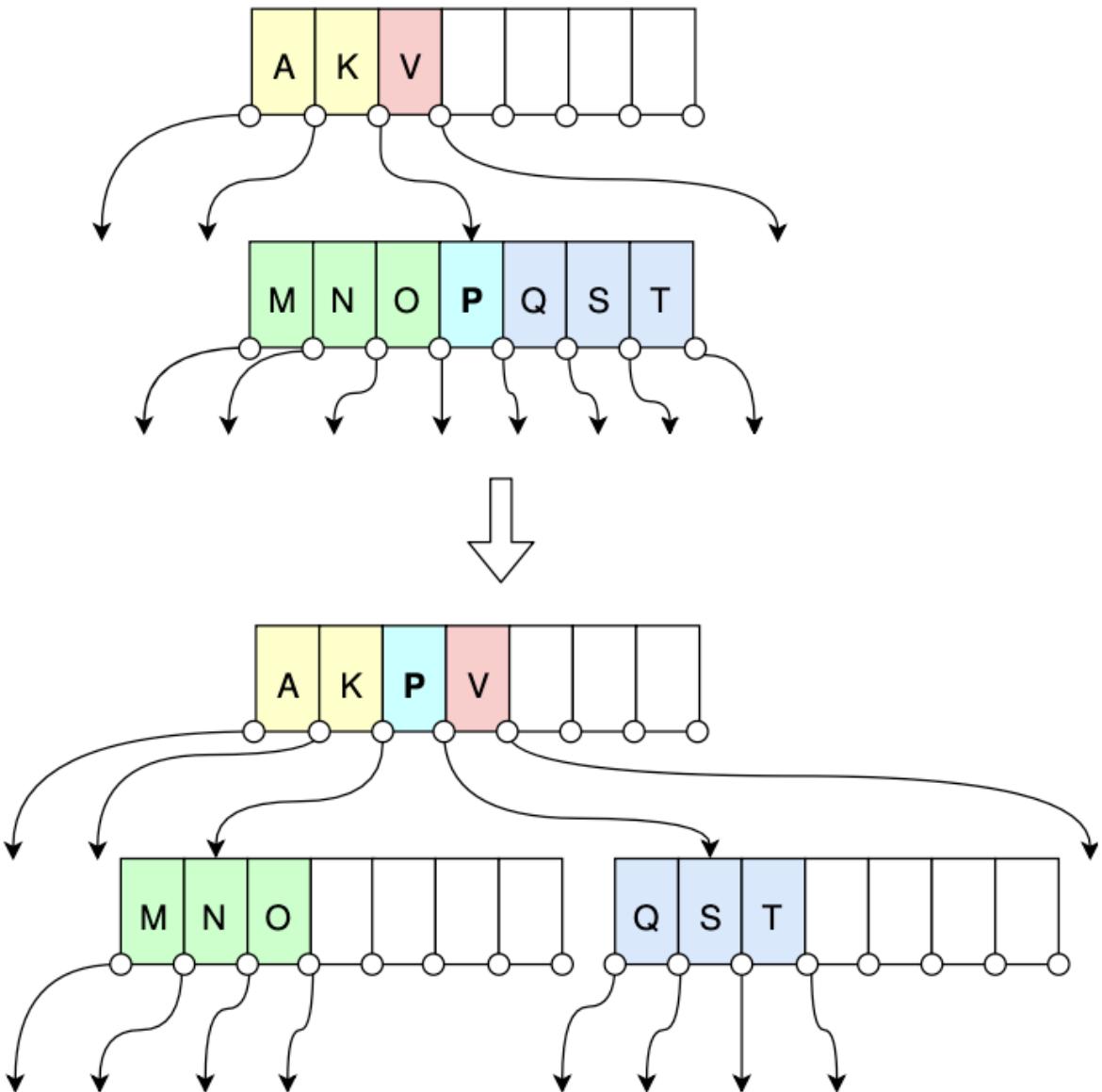
Access time: 50-70ns

# 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  $m=7$

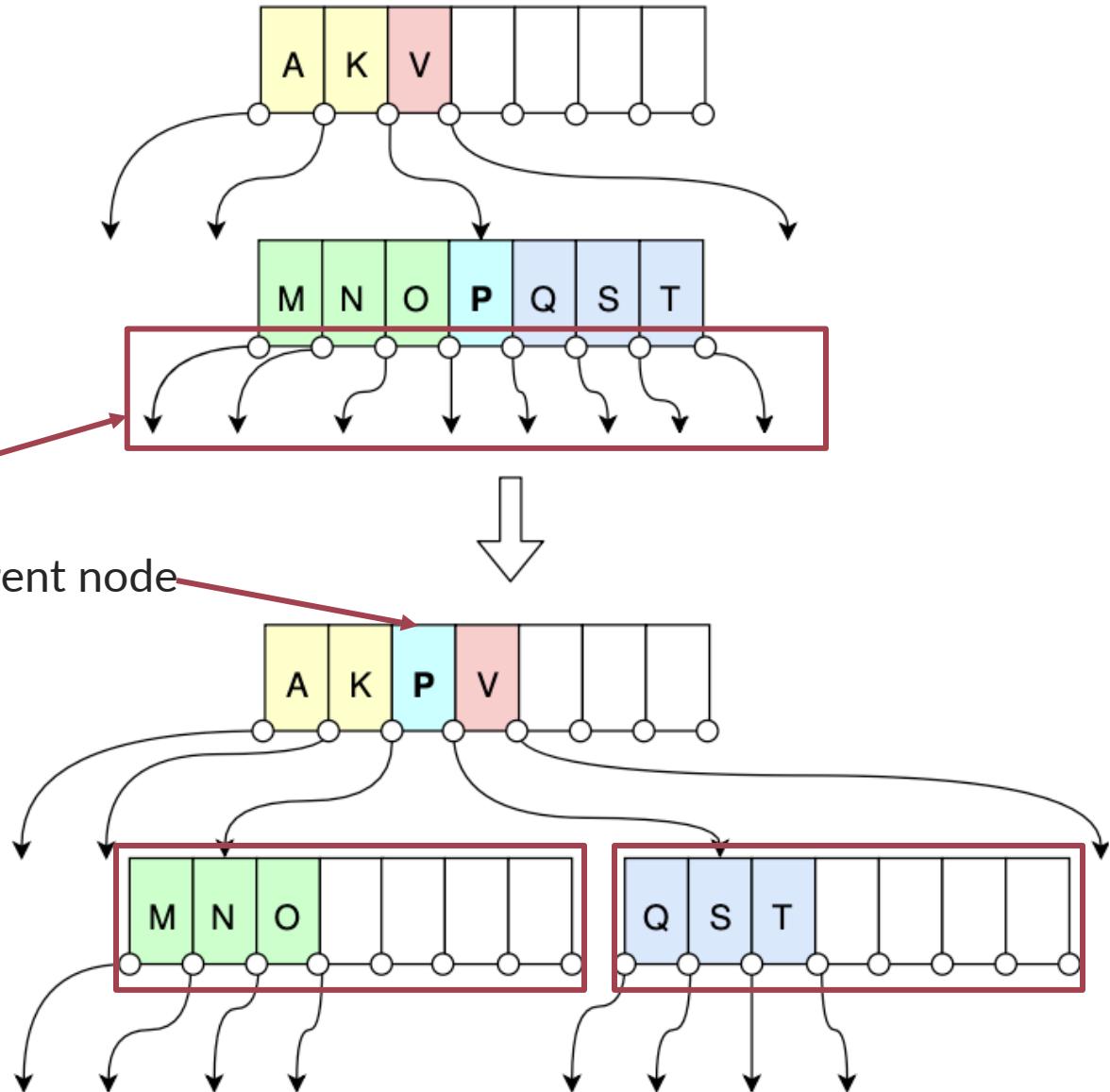


# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$

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

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



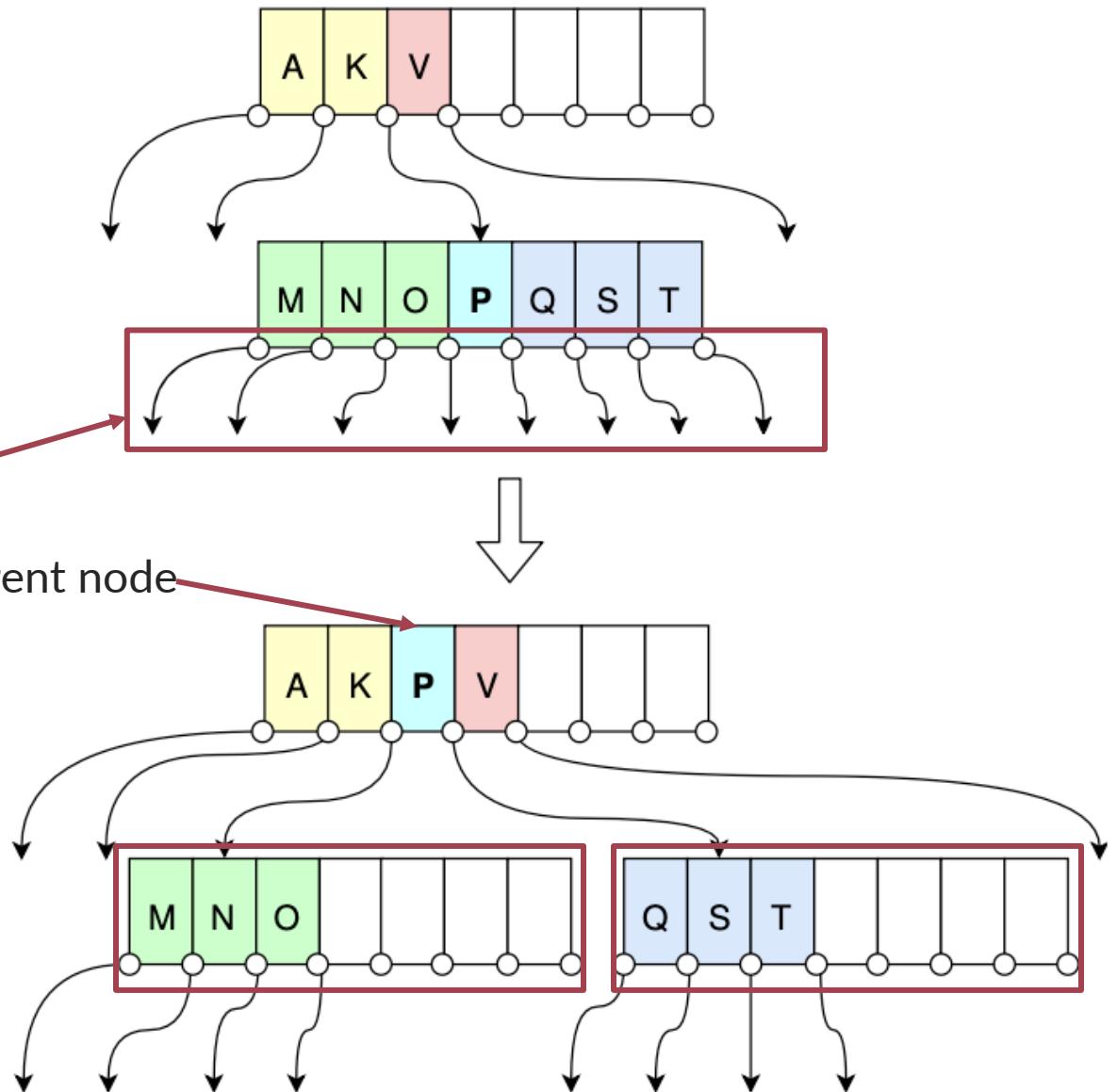
# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$

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

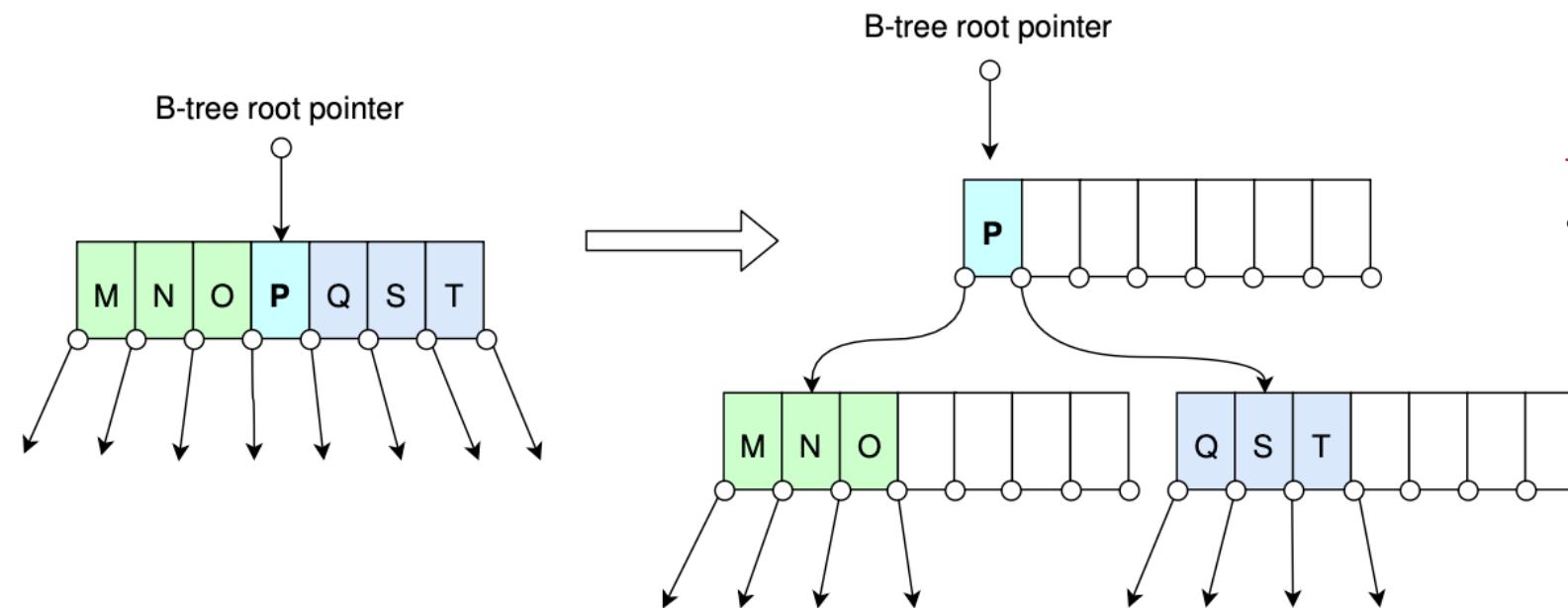
- This node will be split into two nodes
- The pivot key will be elevated into the parent node

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



# B-tree

- Split a node in a B-tree
  - Example: when  $m=7$
- Split the root node of the B-tree



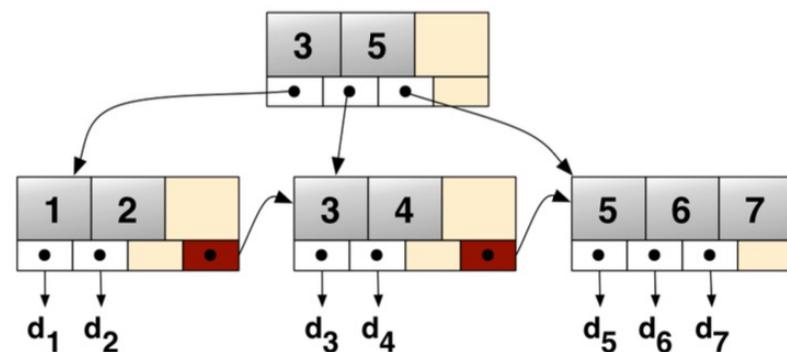
Note that the height of the B-tree is increased by 1

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

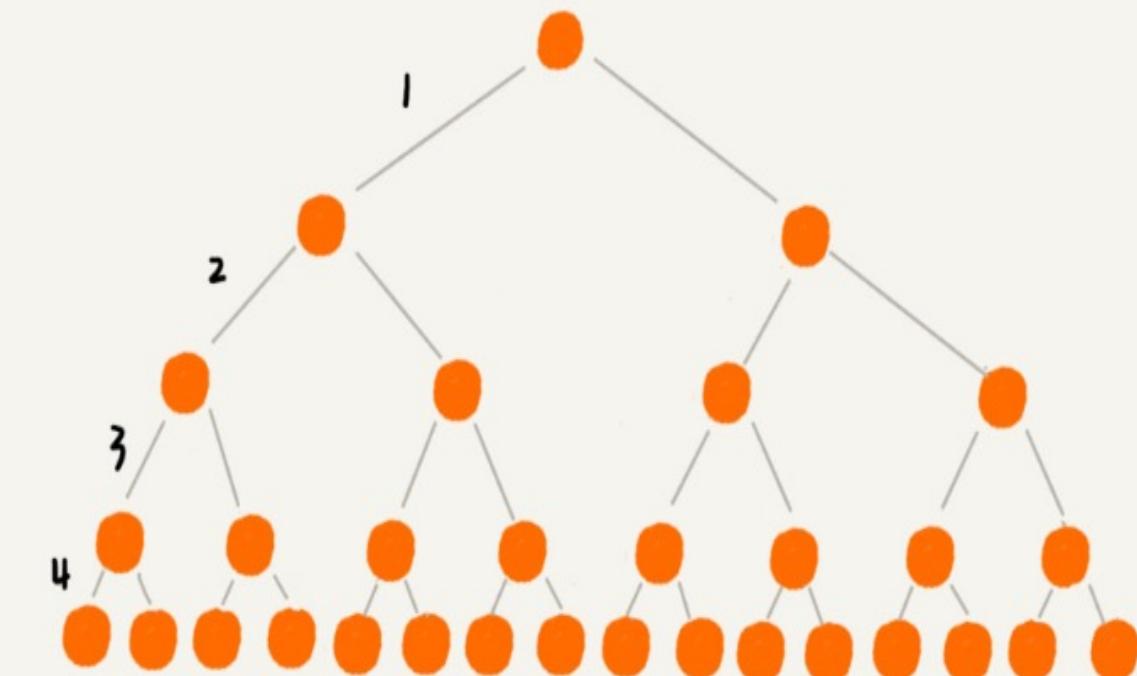
= 叉树

# B+-tree

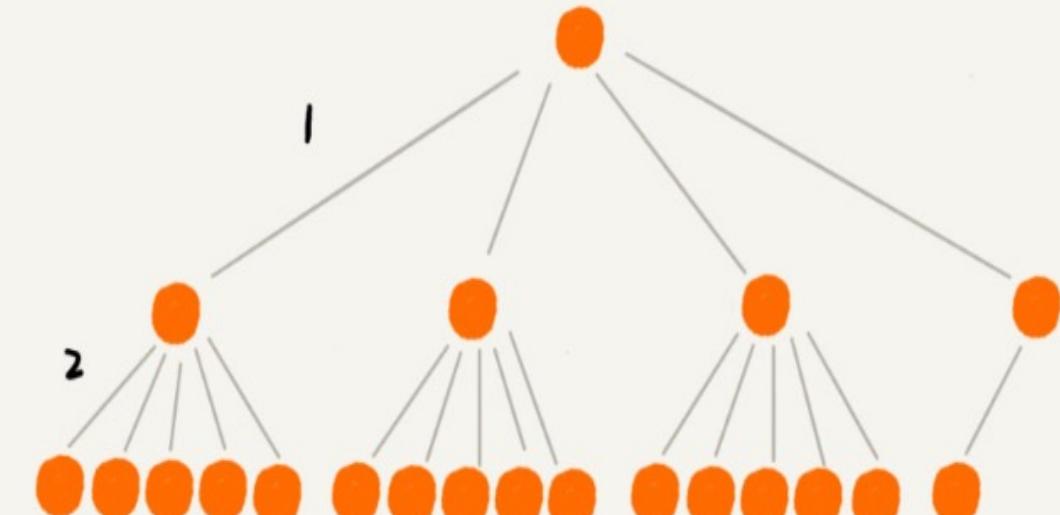
- select \* from user where id = 123 -- easy
- select \* from user where id > 123 and id < 234 ?



- Features of a B+-tree
  - Data stored only in leaves
  - Leaves are linked sequentially

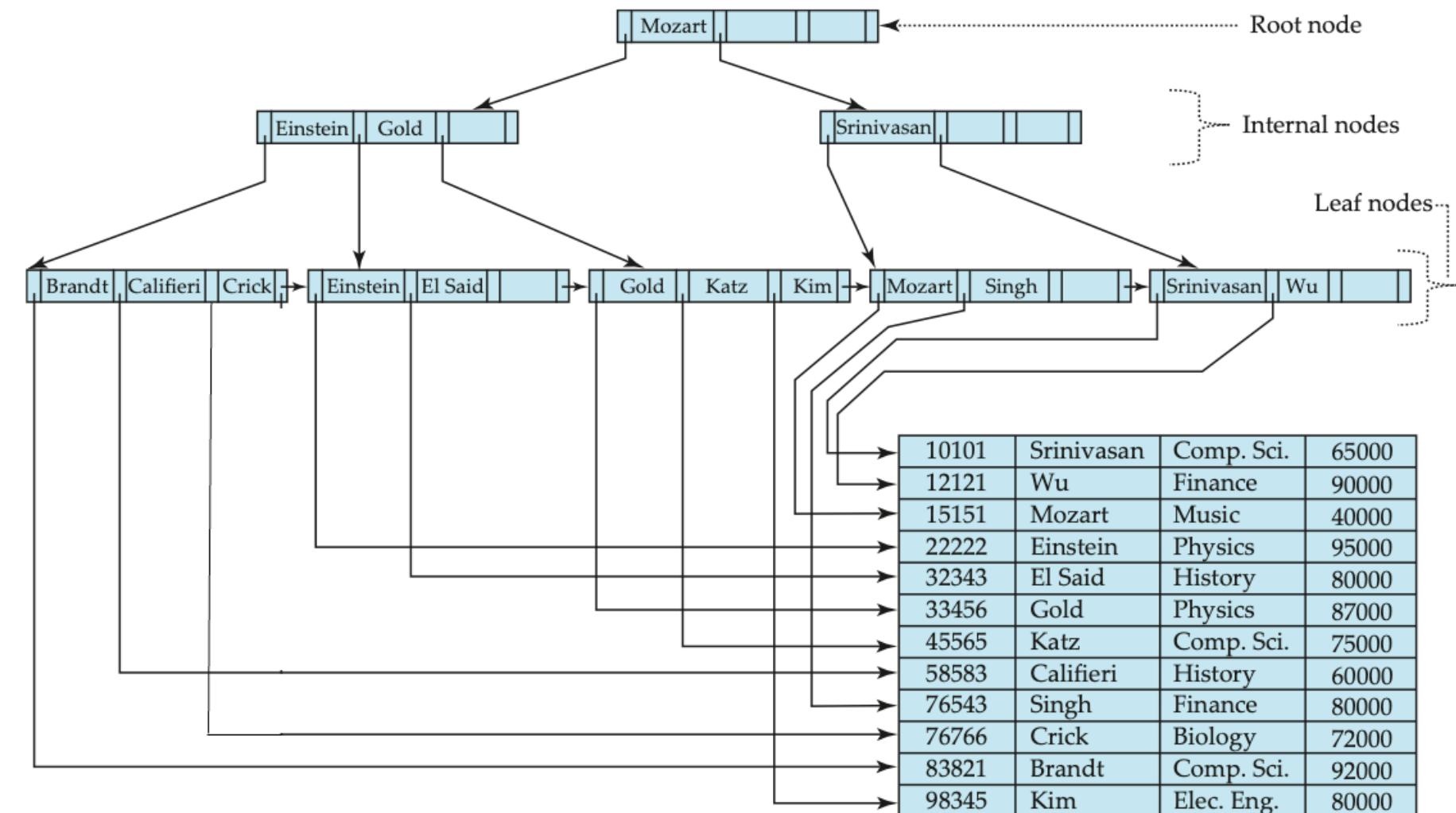


五叉树



# B+-tree

- A complete example of a B+-tree
  - Data stored only in leaves
    - No need to squeeze data into nodes
  - Leaves are linked sequentially
    - Faster table traverse 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
  - insert, update, delete
- 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

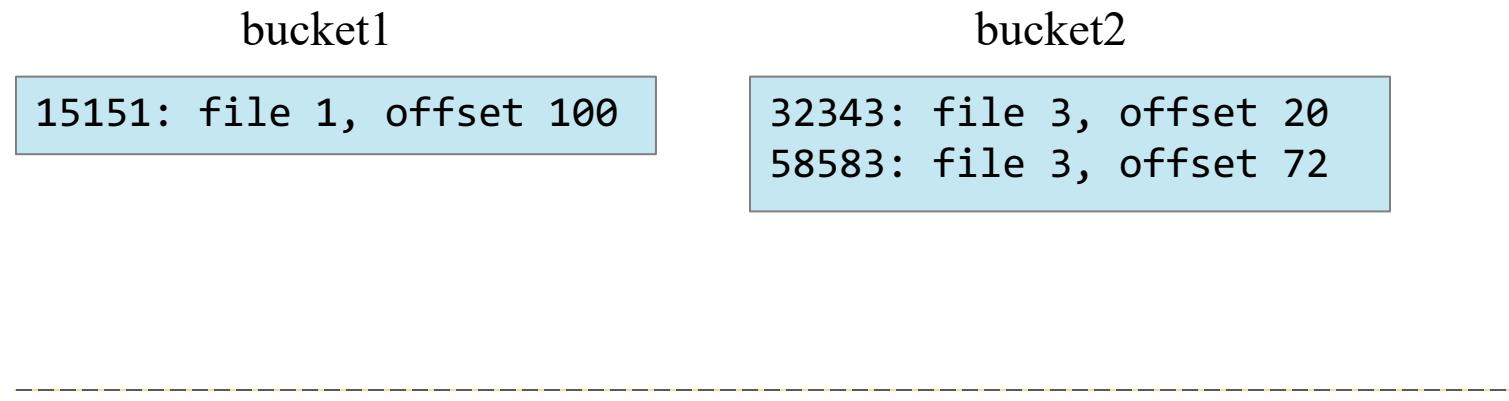
- Be cautious when using indices on a small table
  - Full scan ≠ Bad scheme
  - Index retrieval ≠ Good scheme

# Hashing

- 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 has to be searched sequentially to locate an entry.

# Hashing Index & Hashing File Organization

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



- In a **hash file-organization**, buckets store records

| bucket 0 |  |  |  |  |
|----------|--|--|--|--|
|          |  |  |  |  |
|          |  |  |  |  |
|          |  |  |  |  |
|          |  |  |  |  |

| bucket 1 |        |       |       |  |
|----------|--------|-------|-------|--|
| 15151    | Mozart | Music | 40000 |  |
|          |        |       |       |  |
|          |        |       |       |  |
|          |        |       |       |  |

| bucket 2 |           |         |       |  |
|----------|-----------|---------|-------|--|
| 32343    | El Said   | History | 80000 |  |
| 58583    | Califieri | History | 60000 |  |
|          |           |         |       |  |
|          |           |         |       |  |

| bucket 3 |          |            |       |  |
|----------|----------|------------|-------|--|
| 22222    | Einstein | Physics    | 95000 |  |
| 33456    | Gold     | Physics    | 87000 |  |
| 98345    | Kim      | Elec. Eng. | 80000 |  |
|          |          |            |       |  |

| bucket 4 |       |         |       |  |
|----------|-------|---------|-------|--|
| 12121    | Wu    | Finance | 90000 |  |
| 76543    | Singh | Finance | 80000 |  |
|          |       |         |       |  |
|          |       |         |       |  |

| bucket 5 |       |         |       |  |
|----------|-------|---------|-------|--|
| 76766    | Crick | Biology | 72000 |  |
|          |       |         |       |  |
|          |       |         |       |  |
|          |       |         |       |  |

| bucket 6 |            |            |       |  |
|----------|------------|------------|-------|--|
| 10101    | Srinivasan | Comp. Sci. | 65000 |  |
| 45565    | Katz       | Comp. Sci. | 75000 |  |
| 83821    | Brandt     | Comp. Sci. | 92000 |  |
|          |            |            |       |  |

| bucket 7 |  |  |  |  |
|----------|--|--|--|--|
|          |  |  |  |  |
|          |  |  |  |  |
|          |  |  |  |  |
|          |  |  |  |  |

# How join Works

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

# How join Works

- 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
```

# How join Works

- 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  $t_1$  in the memory
  - 2. Scan the larger table  $t_2$ . For each record  $r$ ,
    - 2.1 Compute the hash value of  $r.\text{join\_attribute}$
    - 2.2 Map to corresponding rows in  $t_1$  using the hash table

# How join Works

- 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.

When there are 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.