COMP90050 ADBS

授课老师：Rita

1 – Indexing

字典目录(帮助加快速度)

DBMS

DBMS admin 管理 index

- **DBMS admin** generally creates **indices** to allow almost direct access to individual items (index ≒ indice)

- DBMS must support:
  - insert/ delete/ modify record 增删改.
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved), or scan a range of records

RDBMS中. primary key. (find特定 record)

primary key 主键

Indexing

- Indexing mechanisms used to speed up access to desired data in a similar way to look up a phone book or dictionary
- **Search Key** - attribute or set of attributes used to look up records/ rows in a system like an ID of a person (The ID will take us to get the entire record)
- An **index file** consists of records (called index entries) of the form search-key, pointer to where data is
- Index files are typically much smaller than the original data files and many parts of it are already in main memory (main memory is faster than disk)

Index 自身加速部份 → index 能直接帮助我们指定一个查找位置。

Indexing makes Disk Access faster through

how

- records with a **specified value** in the attribute accessed with minimal disk accesses (e.g., Student ID = 101, the index file tells us which data block in disk to go)

  index file 存着一些 specified value 之后，直接找到存储位置。

- records with an attribute value falling in a **specified range** of values can be retrieved with a single seek and then consecutive ~~sequential reads~~ 顺序 (e.g., ID:100- 200, find where is the first record (single seek of starting block), then read sequentially)

  一些 range

  index file 先定位置 → 然后顺序找

Note: not all DB stores sequential data sequentially, so this method may not useful for some DB

(有些存的，specified range 就没用了)

Criteria to choose index – always tradeoff

- Insertion time to index is also important
- Deletion time is important as well
- No big index rearrangement after insertion and deletion
- Space overhead needs to be considered for the index itself
- No single indexing technique is the best. Rather, each technique is best suited to particular applications.

Types of indices based on search keys

- **Ordered indices** – Search keys are stored in some order
  - Clustering index / **primary index**
  - Non- clustering index / **secondary index**
- **Hash indices** – Search keys are distributed
- **B+tree**
- **Bitmap index**
- **Quadtree (k-d tree)**
- **R-tree**

data is ordered based on salary ⇒ index is also ordered based on the same key. ⇒ clustered index

Clustering Index / Primary Index

in a sequentially ordered file, the index whose search key specifies the sequential order of the file

(The search key of a primary index is usually but not necessarily the primary key)

search key 通常 primary key 但不必须

attribute

计: search key 按 name 来排 → 有首字母重复的 → 开始增大 → W1 W2 W3 Z1 Z2 Z3 Z4

Id 按 order 排 从小到大

eg☆ 100 million 数据是 CS 的老师 要搜索所有 rows one by one → 用不上 ID (若 index 在 ID 上 就没用了) (若 index 在 subject 上呢?)

No index / B+ tree

Id	name	favorite subj	salary	
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	

定位数据在 disk 上的位置 → disk access time 会缩短

Index在ID上：要是选择去retrieve every row ⇒ query optimizer会选择 non-index query path ⇒ usiy index is overhead

Lan not provide any benefit 但 but the cost of index is minimum ⇒ just stay in main memory .

Index在subject上：

Non-clustering Index / Secondary Index

data is not ordered based on that column ⇒ it would be non-clustered index.

an index whose search key specifies an order different from the sequential order of the file 不按顺序存储
(Secondary indices improve the performance of queries that use keys other than the search key of the clustering index.)

ID	name	dep
1	Jane	SCI
2	John	Biology

ID	position	Salary
1	Lecturer	75000
1	RA	40000
2	SL	82000

Find the name of instructors with salary 75000

① No index

② Btree index on Salary.

③ 快速定位到大概的区域 ⇒ 扫描

☆answer: 若每个人的 Salary 都好 80000 ⇒ performance are the same

☆比如每个人 Salary 70 ⇒ 查询条件是选出 Salary 70的 ⇒ 一样 performance ☆

☆Conclusion 如果是那种需要遍历 retrieve every row of table 的查询 ⇒ 有无 index 是一样的.
or the row satisfy the condition

⇒ we have to retrieve the whole table then do the join.

⇒ it will not reduce any number of page! ☆

☆这个 index 是在 join 前的 ☆

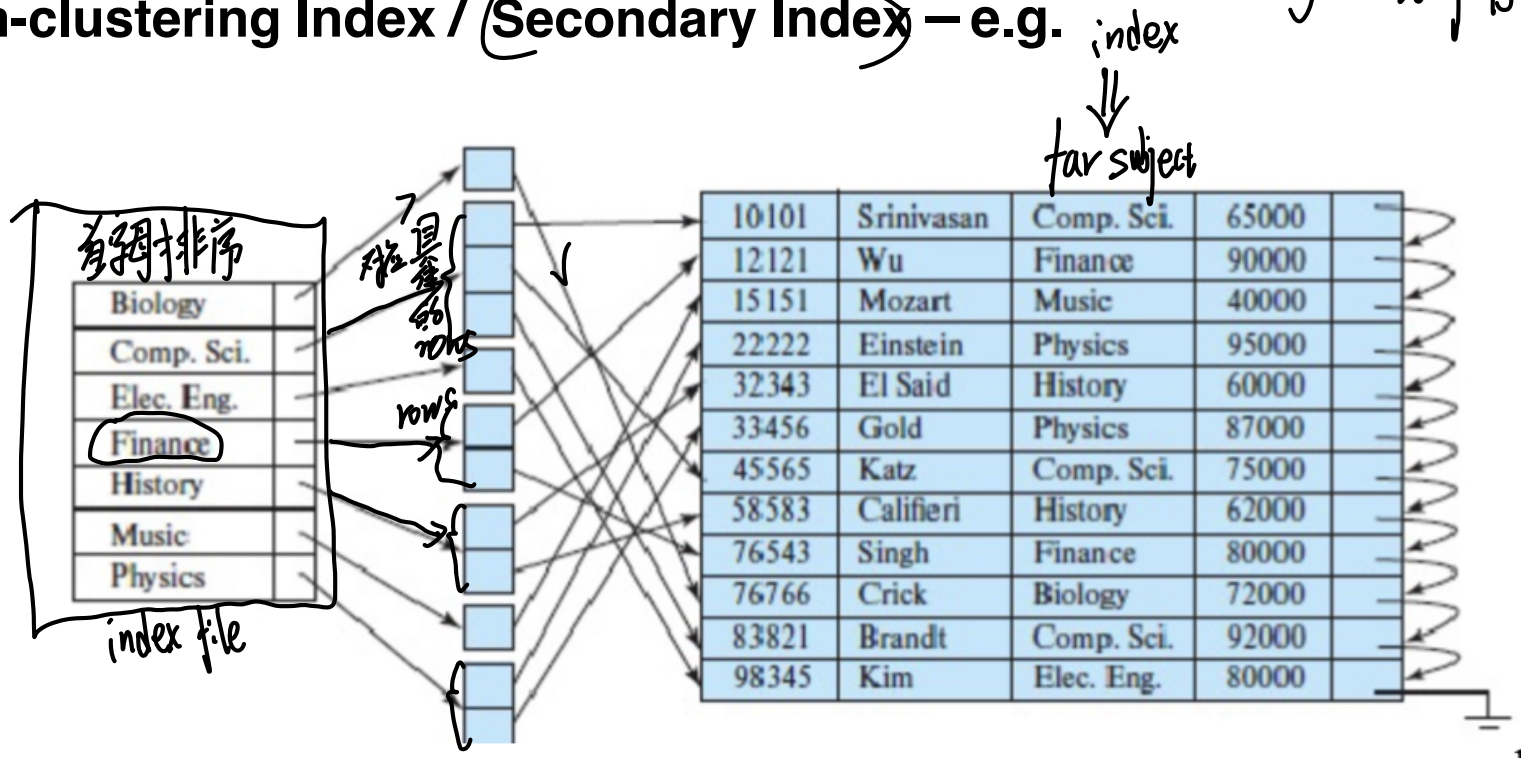
join 后无法使用, 先 index ⇒ 再 join
index

先 join ⇒ 生成的 table 上没 index 没用

non

不是按顺序排列 → 而是找个 search key → 按 search key 的顺序排.

Non-clustering Index / Secondary Index – e.g.



Hash indices search keys are distributed hopefully uniformly across "buckets" using a "function"

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure. Order is not important.
- Given a key, the aim is to find the related record on file in one shot which is important.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search- key values from the set of all possible values.
Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search- key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search- key.

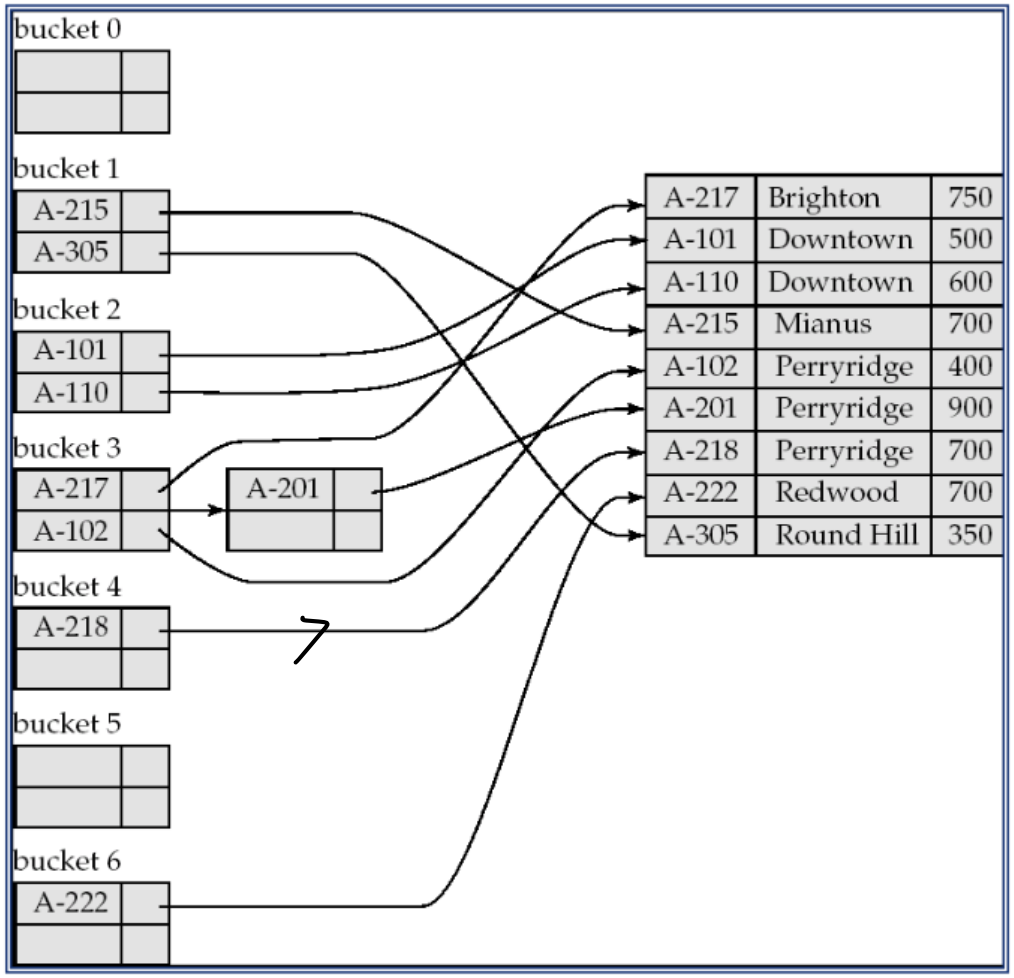
Hash index – e.g.

hash function: hash(x)=y

bucket(桶)：0-6共7桶，称桶为

Hash(A-215)=bucket 1 =>row

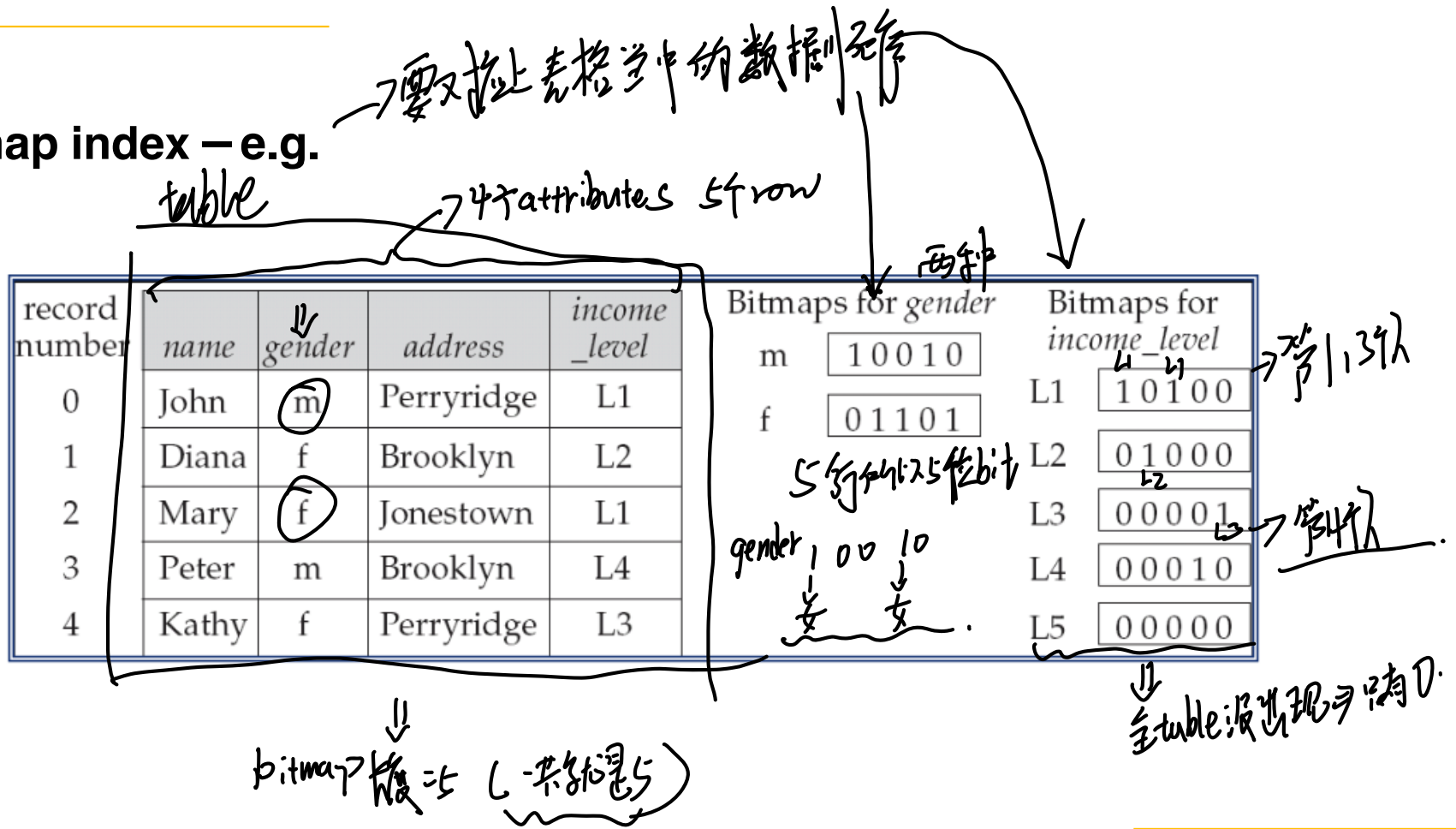
Hash(A-218)=4=>bucket 4 =>row



Bitmap index

- Records in a relation are assumed to be numbered sequentially from, say 0
- Applicable on attributes that take on **a relatively small number of distinct Values**
  - e.g. gender, country, state, ...
  - e.g. income- level (income broken up into a small number of levels such as 0-9999, 10000- 19999, 20000- 50000, 50000- infinity)
- A bitmap is simply an array of bits
- In its simplest form, a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
  - Used for business analysis, where rather than individual records say how much of one type exists is the query/ important

Bitmap index – e.g.



record number	name	gender	address	income_level
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

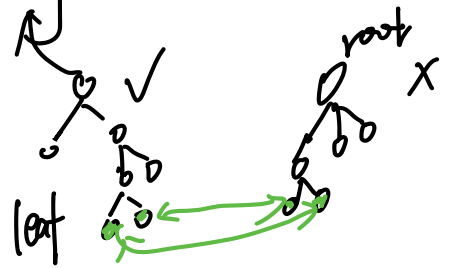
Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Binary tree: each node has at most 2 children



B+tree

Why?

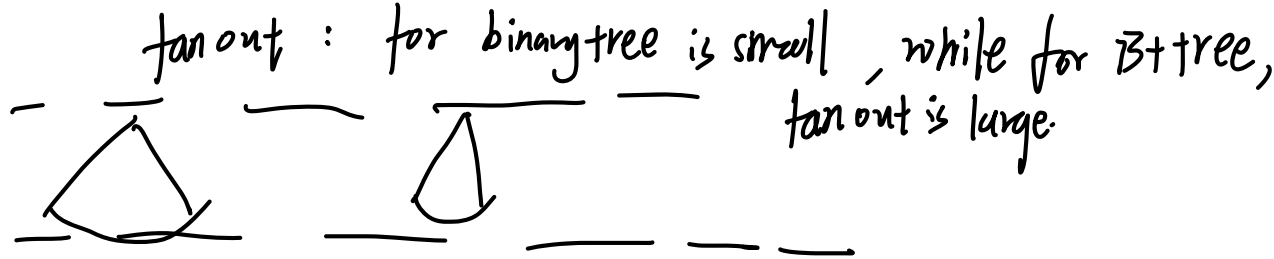
- Keeping files in order for fast search ultimately degrades as file grows, since many overflow blocks get created.
- So binary search on ordered files cannot be done.
- Periodic reorganization of entire file is required to achieve this.

按顺序存数据不好：insert 和 delete 都会让后边数据进行 reorganized，

⇒ 增加了移动的东西太多 ☆☆

With B+ tree, we do not need to reorganize the entire file in the face of insertions and deletions, instead we just need small local changes.

binary search 的前提条件：order ⇒ 整个过程无法进行



B+tree Definition

- It is similar to a binary tree in concept but with a fan out that is defined through a number n
- All paths from root to leaf are of the same length (depth)
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- 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 (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

B+tree Typical Node

- Typical node

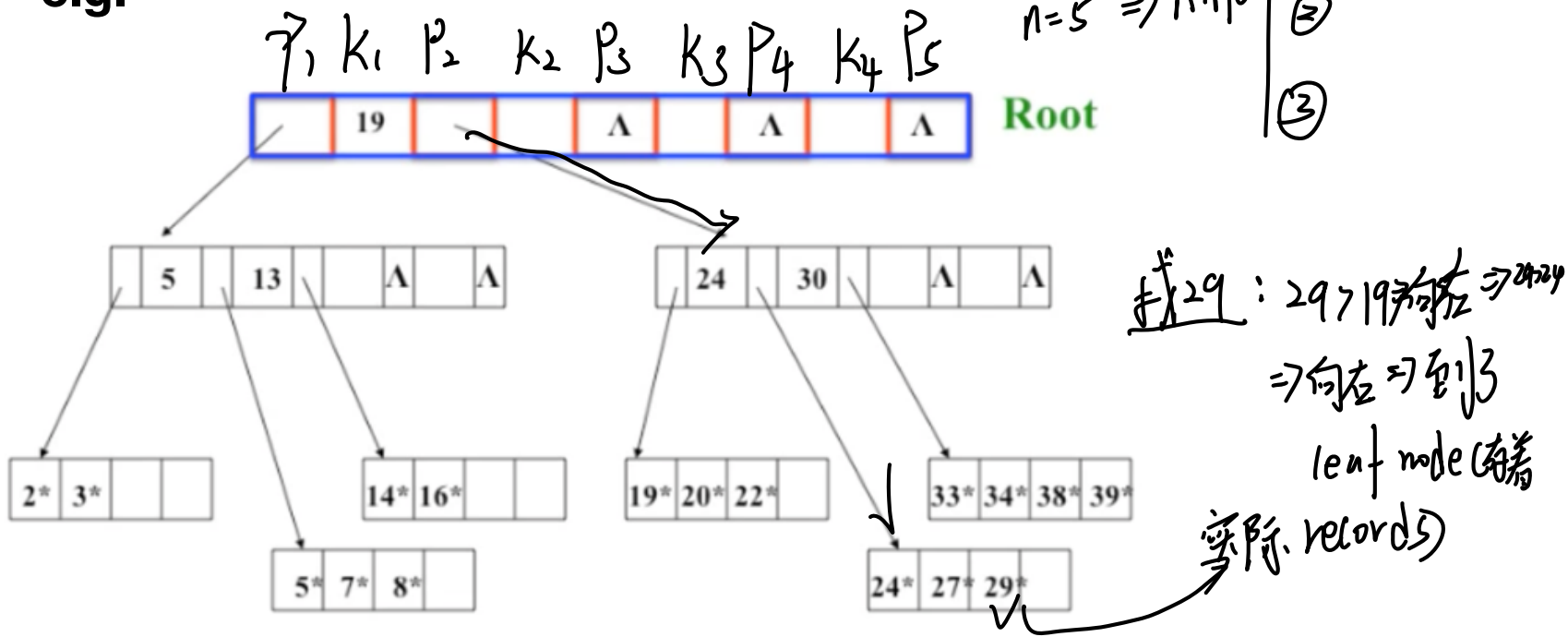
- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

NOTE: Most of the higher level nodes of a B+ tree would be in main memory already!
(leaf node will be in disk)

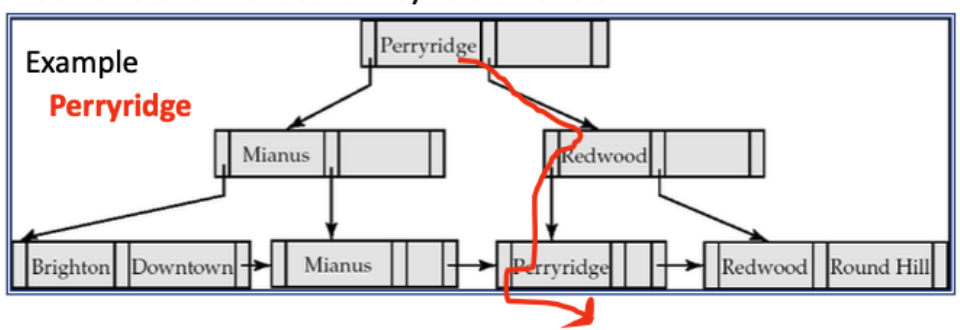
B+tree – e.g.



Run a query in B+tree

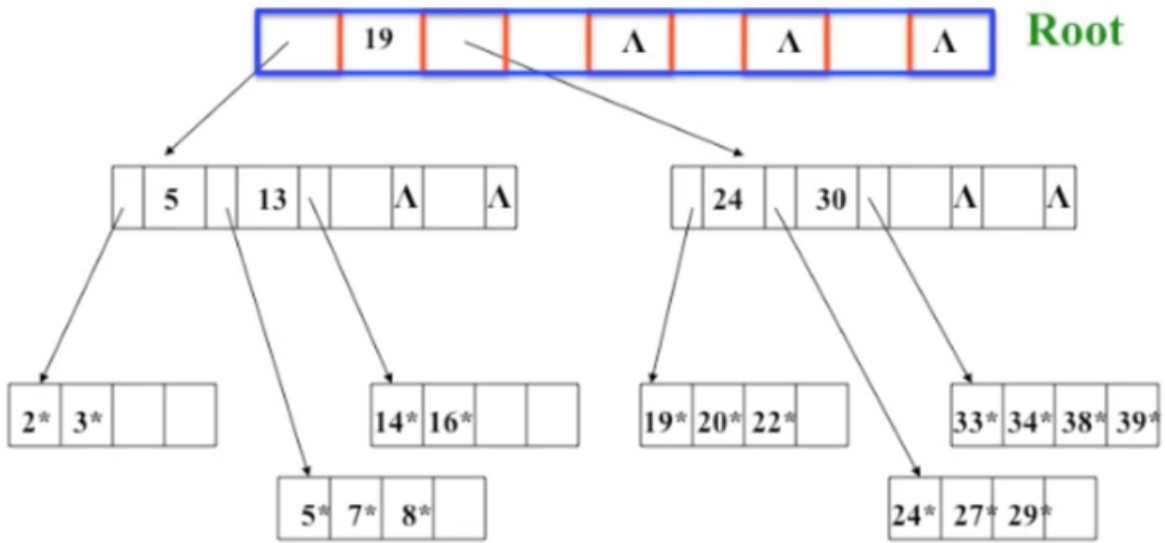
- Finding all records with a search-key value of k.
  1. *N=root initially* 从root开始找含node上的值向右找，otherwise向左找
  2. Repeat
     1. Examine N for the smallest search-key value > k.
     2. If such a value exists, assume it is K_i. Then set $N = P_i$
     3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$. Follow pointer.

     Until N is a leaf node
  3. If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket.
  4. Else no record with search-key value k exists.



Run a query in B+tree – e.g., find 28/29

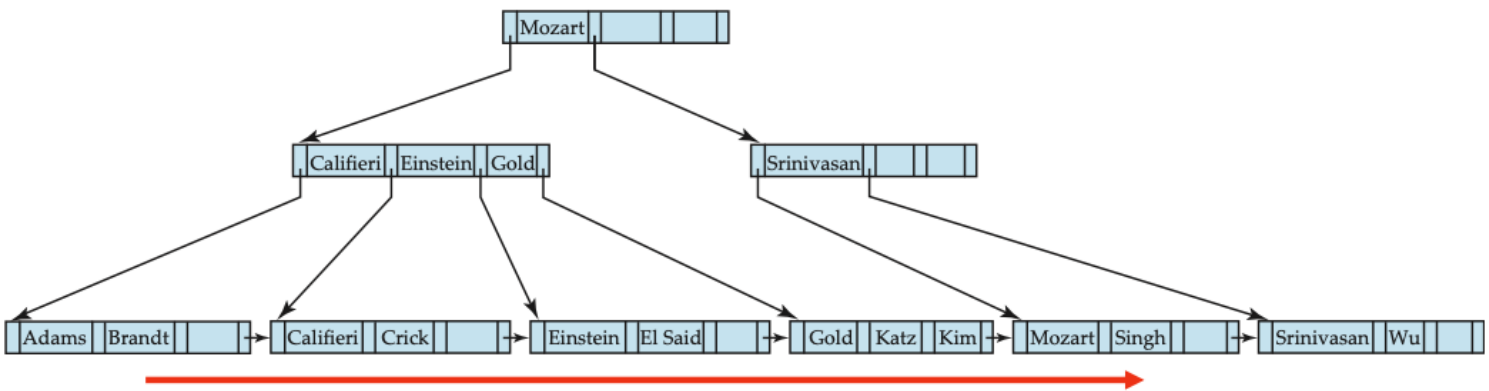
找28 19向右→24向右=找到=返回 not found



Run range queries in B+tree

Range queries find all records with search key values in a given range

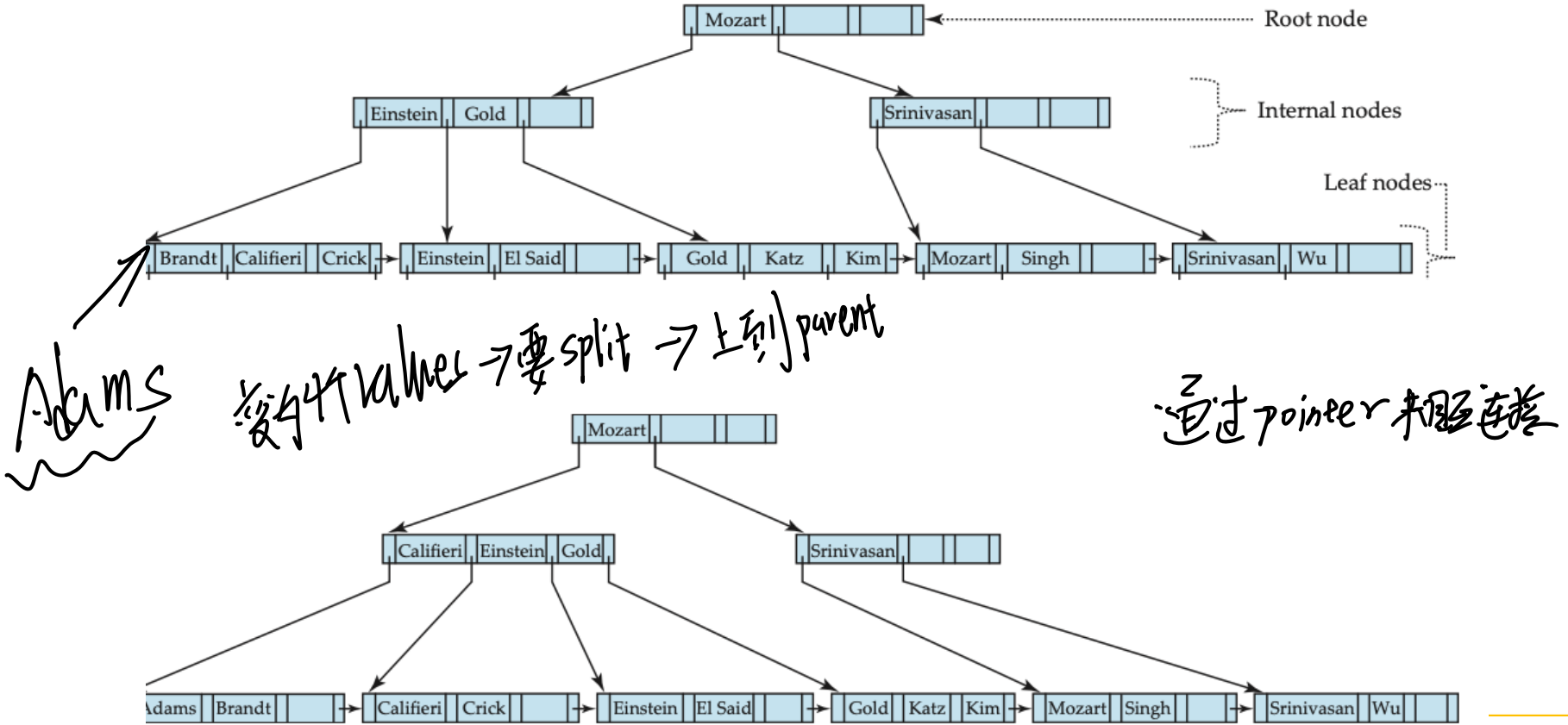
e.g., from Brandt to Singh



Basic Rules for this Tree (n = 4)

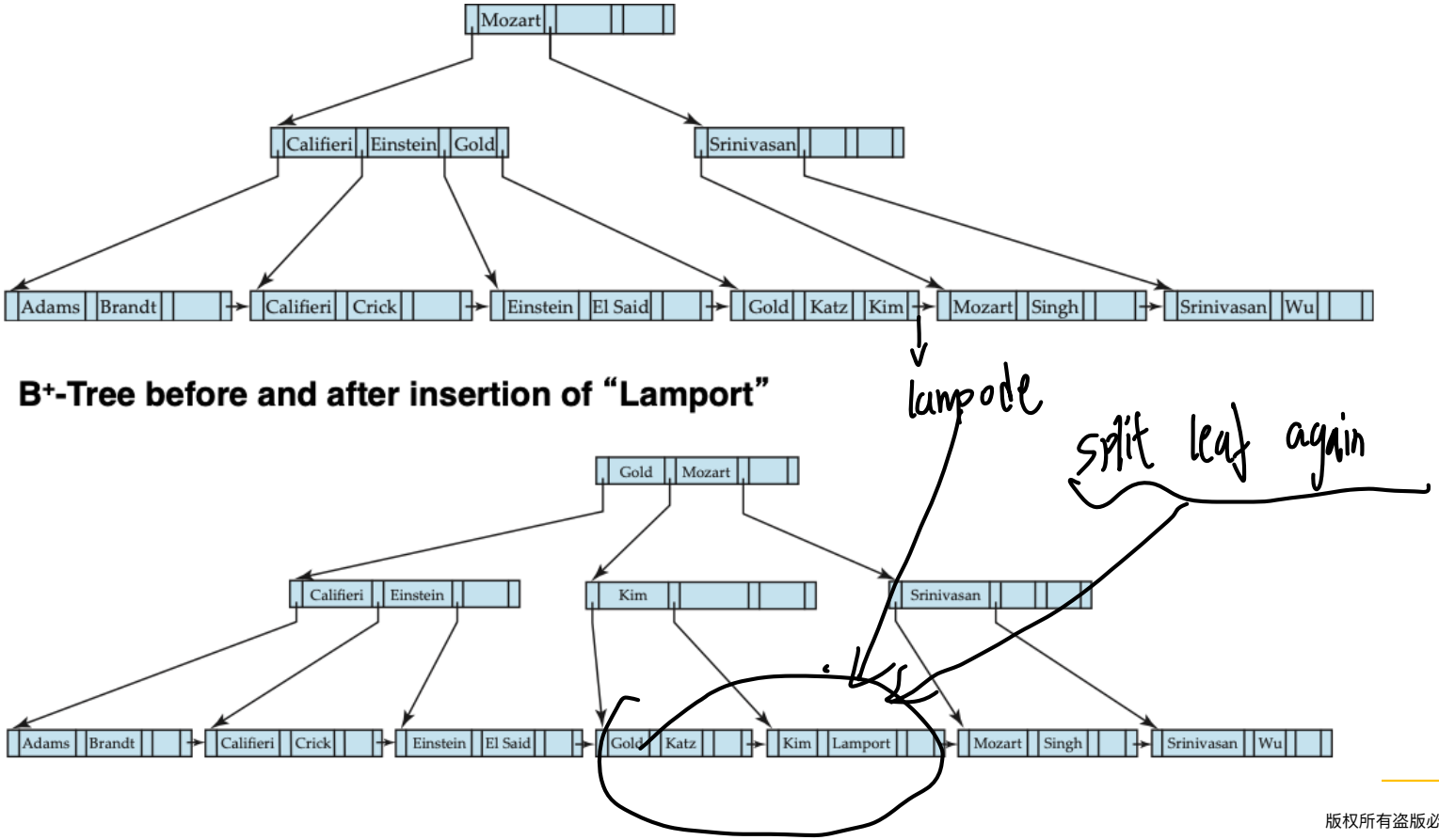
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

B+tree - Insertion (Type 1 – Parent node has room)

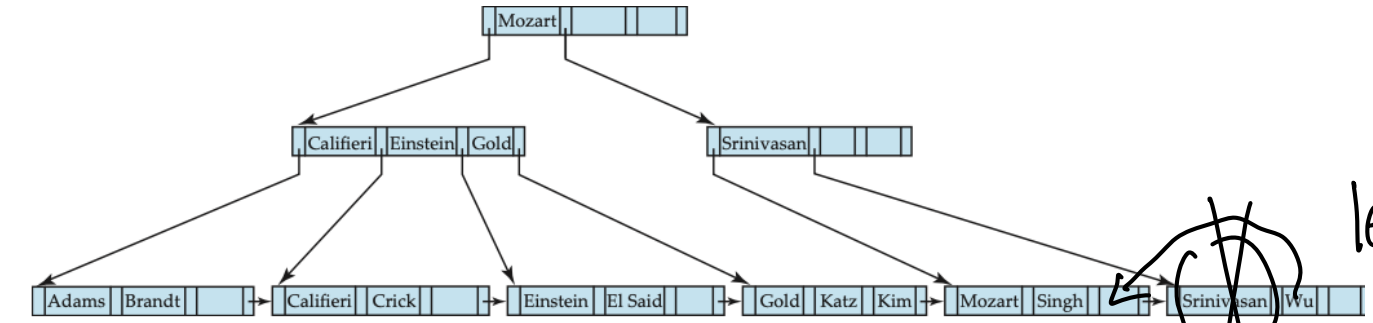


B+-Tree before and after insertion of "Adams"

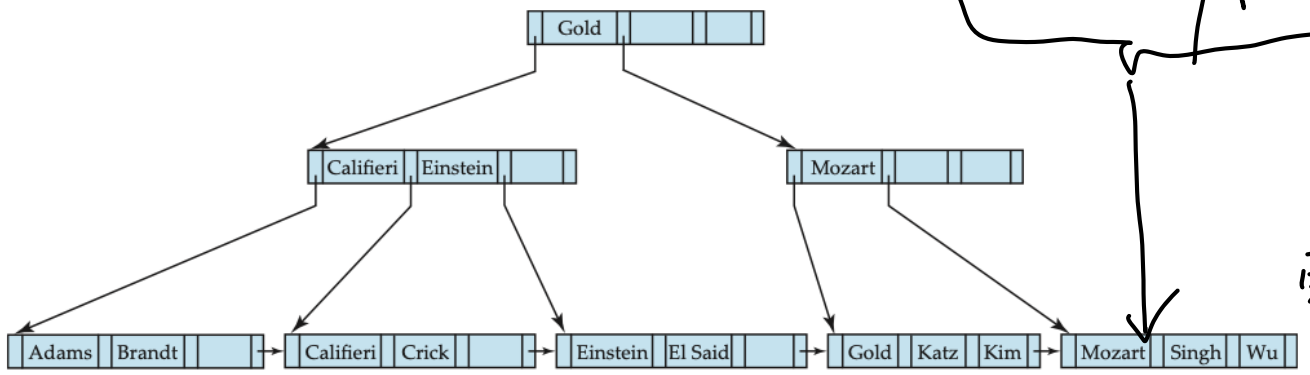
B+tree - Insertion (Type 2 – Parent node does not have enough room)



B+tree - Deletion (Type 1 - merge)



Before and after deleting "Srinivasan"



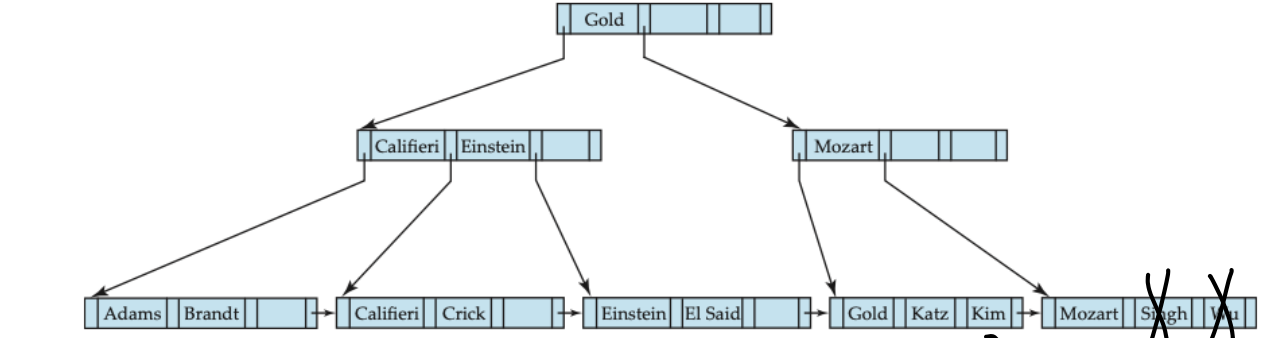
Deleting "Srinivasan" causes **merging** of under-full leaves

leaf not 要有 $C\left\lceil \frac{n-1}{2} \right\rceil, n-1$ 个 value.

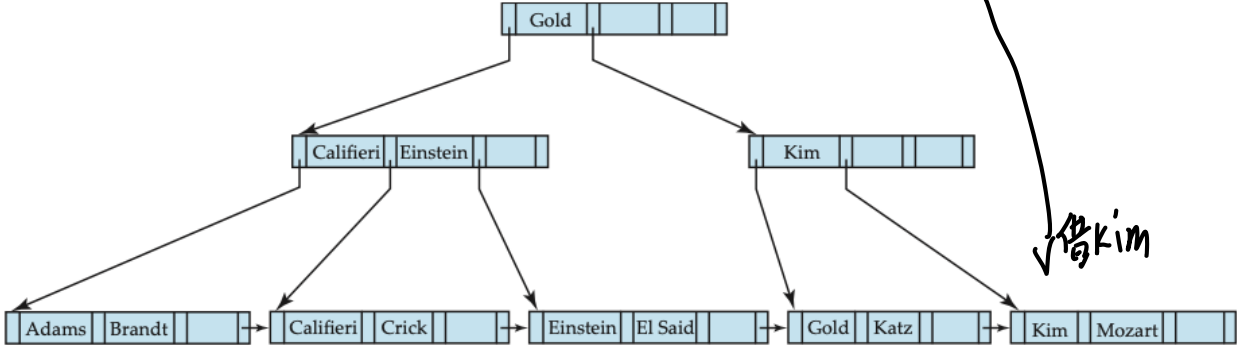
只有1个

要 merge = Mozart 放到第一个

B+tree - Deletion (Type 2 - borrow)



Before and after deleting “Singh” and “Wu”



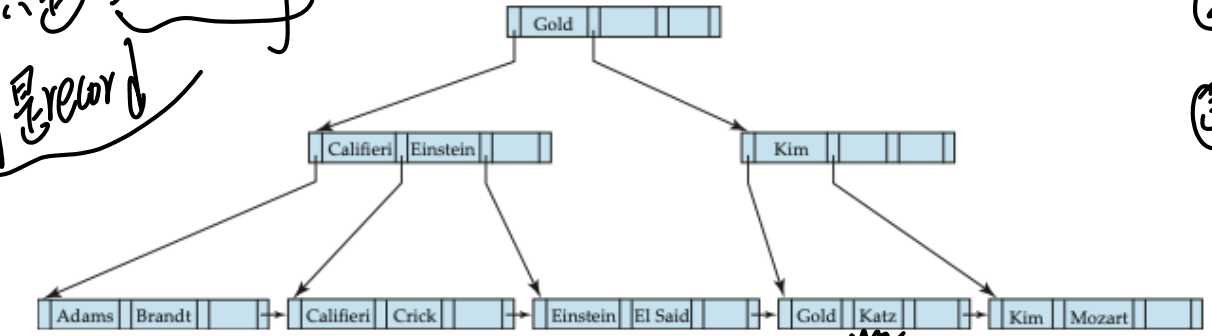
Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling

Search-key value in the parent changes as a result

B+tree - Deletion (Type 3 – one level is removed)

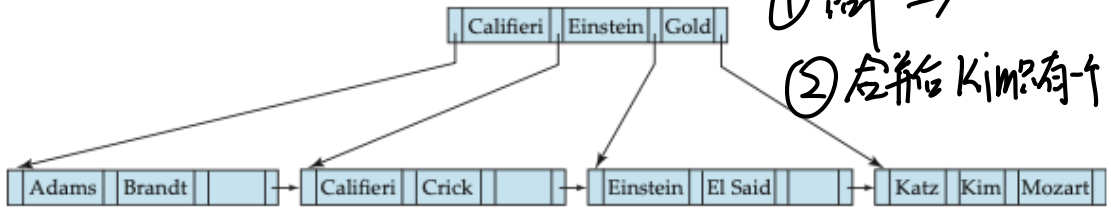
node上不是search key
叶子上才是record

n=4
① leaf node 2-3个value
② 中间node 2-4个children
③ root至少有2个children.



Before and after deletion of "Gold"

① 落单 ⇒ Katz merge with left



② 合并后Kim没有child ⇒ Kim和旁边的merge.
{Califieri, Einstein, Kim}
Kim merged with left

- Node with Gold and Katz became underful, and was **merged with its sibling** ③
- **Parent node becomes underfull**, and is merged with its sibling
- Value separating two nodes (at the parent) is **pulled down when merging**
- **Root node** then has only one child, and is **deleted**

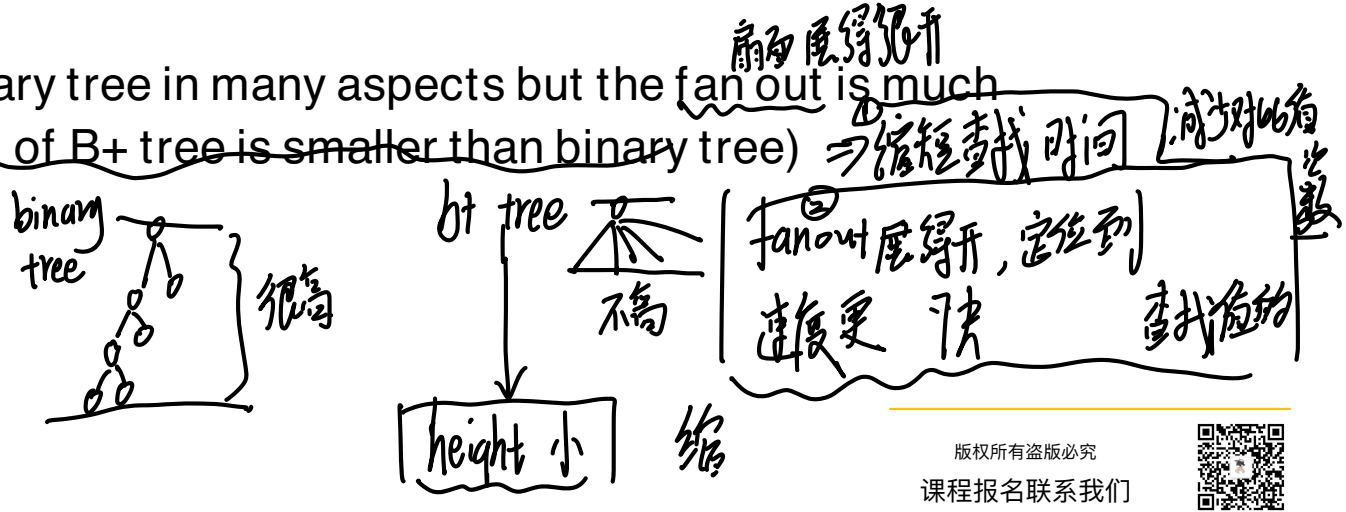
B+tree – File organization

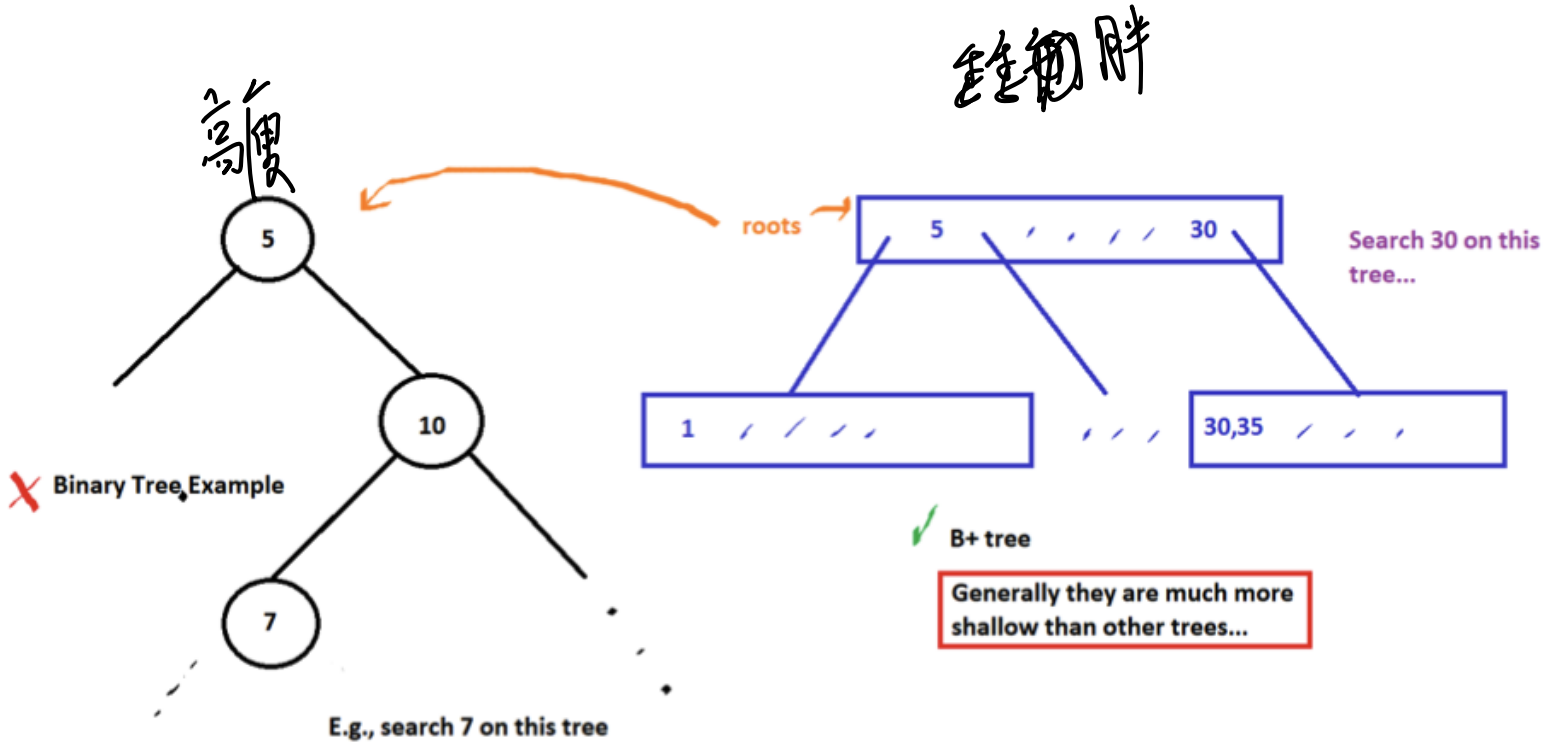
- Leaf nodes in a B^+-tree file organization store records, instead of pointers to children
- Helps keep data records clustered (ordered) even when there are insertions/deletions/updates
- Insertion and deletion or records are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+tree

Advantages

- Automatically reorganizes itself with small, local changes in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.
- Similar to Binary tree in many aspects but the fan out is much higher (height of B+ tree is smaller than binary tree)





B+tree

Disdvantages

Extra insertion and deletion over head and space over head (reorganisation)

Advantages of B+- trees outweigh disadvantages for DBMSs
(B+ trees are used extensively)

Considering location data (spatial data)...

Unlike things we can access by names, ids, there is a lot of data that exists, and increasingly that requires special indexing

For example, **spatial data requires more complex computations** for accessing data, e.g., intersections of objects in space

There is no trivial way to sort items which is a key issue,
e.g., range query on a simple set of items
e.g., a Nearest Neighbour query

index 就 Q-dtree 对数据点

Quadtree (for location data - points)

Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.

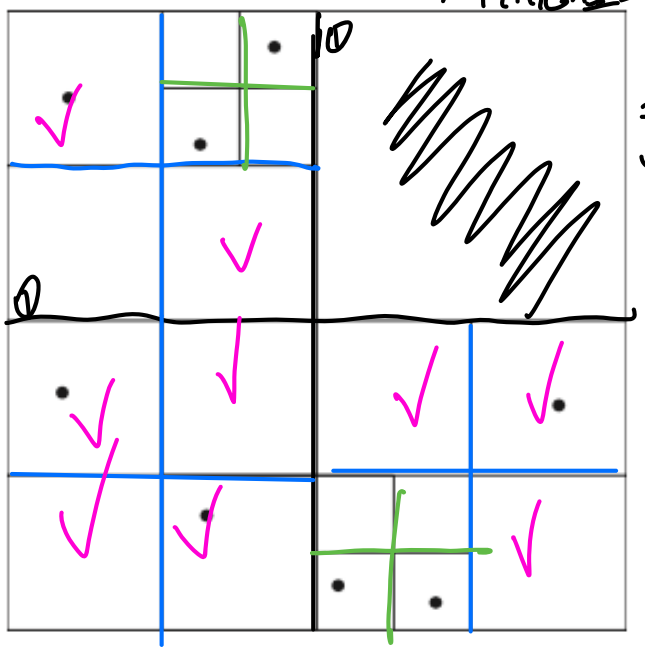
Each division happens with respect to a rule based on data type.

Each non-leaf nodes divides its region into four equal sized quadrants

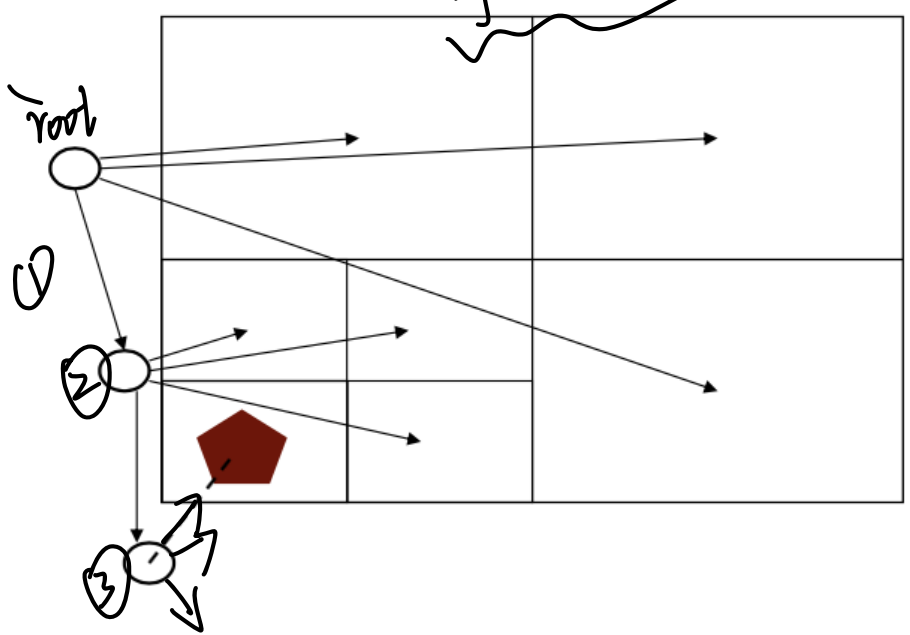
Thus each such node has four child nodes corresponding to the four quadrants and division continues recursively until a stopping condition

Quadtree – e.g.,

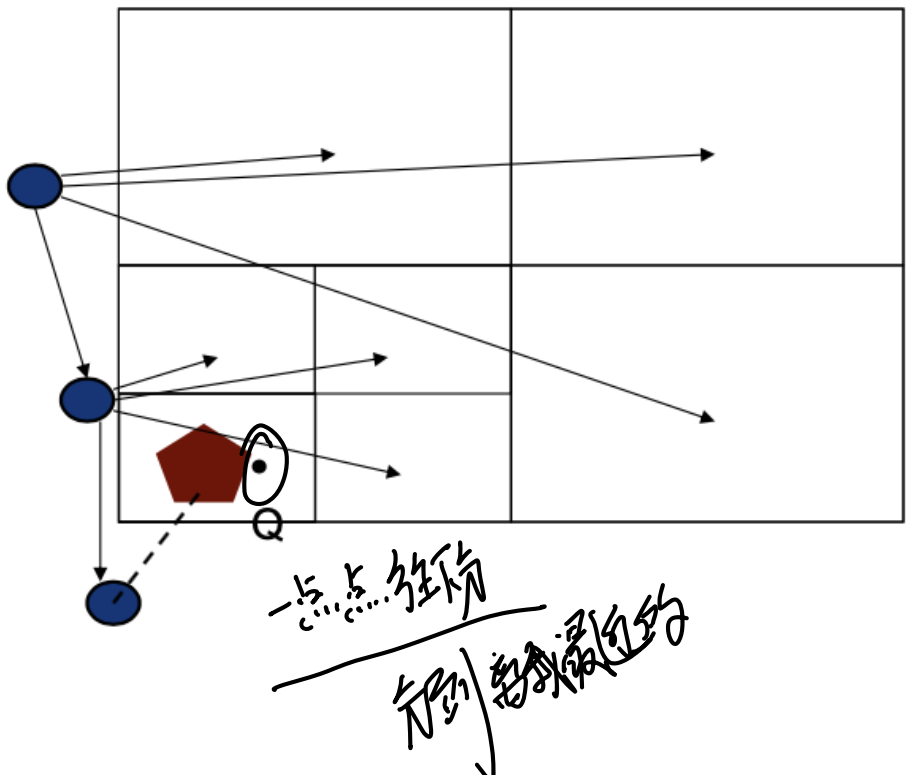
Example: Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example below)



Quadtree – where is the tree?



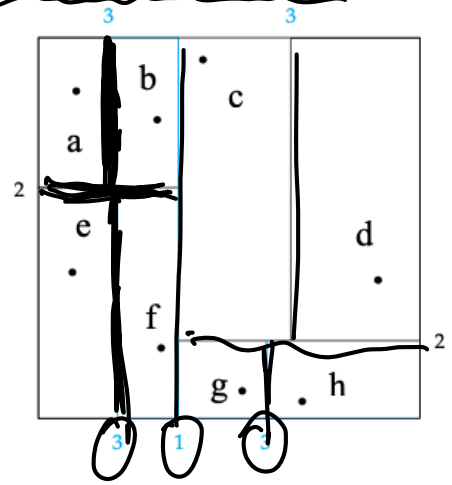
Quadtree – e.g., run NN query

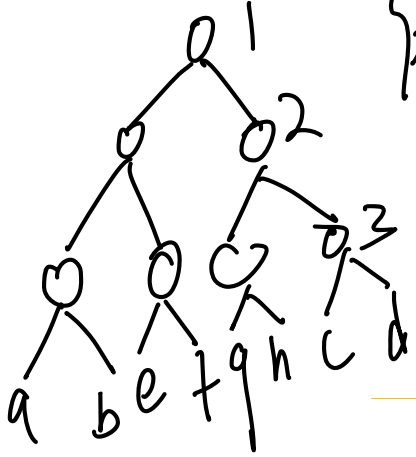


2022 S1 (k-d tree)

Question 5: [4 Marks]

Given the k-d tree below with point data, where black labeled dots represent spatial coordinate data, and the rectangular area is divided into regions with the division order given with numbers: draw the associated k-d tree as a tree structure with leaves labeled as the data labels given below. Assume left subarea of a division goes to a left subtree, and lower subarea of a division also goes to a left subtree. The figure for the k-tree follows:





R-tree (for location data – rectangles & polygens)

R-trees are an N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.

Supported in many modern database systems, along with variants like R^+-trees and R*-trees.

Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.

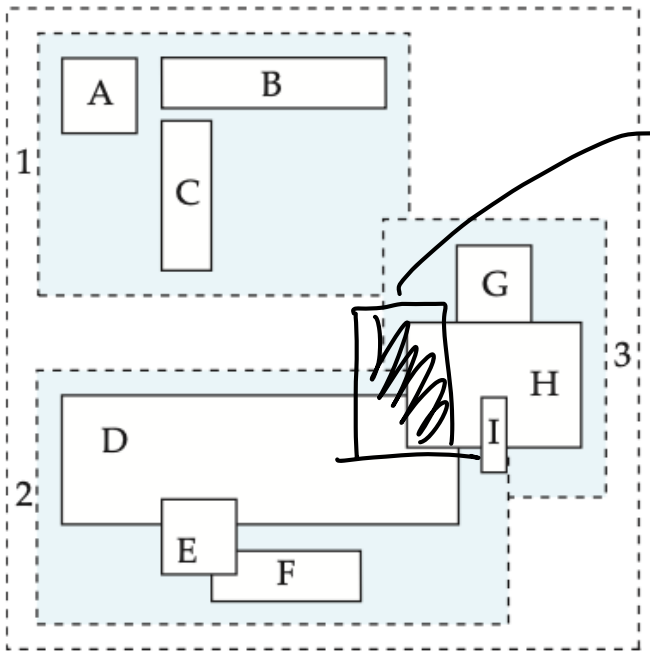
Will consider only the two-dimensional case ($N = 2$)

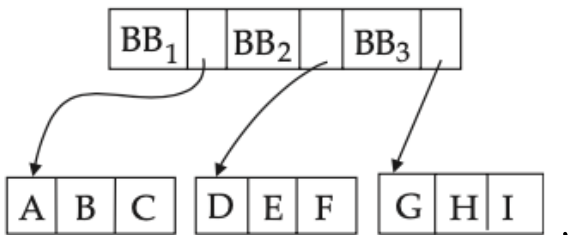
- generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N

Bounding boxes of children of a node are allowed to overlap

Note: A **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node

R-tree – e.g.,





Search in R-tree

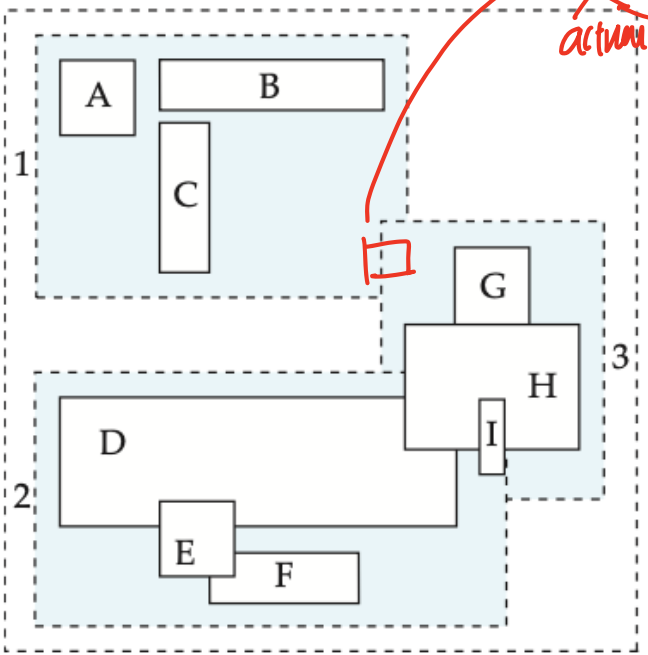
To find data items intersecting a given query point/region, do the following, starting from the root node:

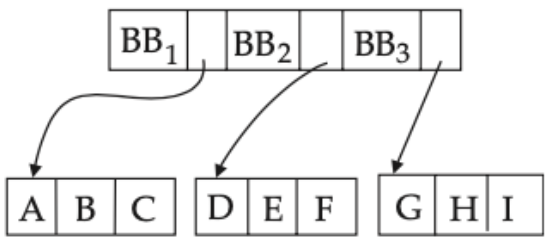
- If the node is a leaf node, output the data items whose keys intersect the given query point/region.
- Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child.

Can be very inefficient in worst case since multiple paths may need to be searched due to overlaps, but works acceptably in practice.

there might be multiple overlaping bounding boxes ⇒ which may lead to explore multiple parts or like going through branches of this tree to find the actual result. 举例看下面:

Search in R-tree – e.g.,





my query is that location however.

there's o
no objective here but my query actual

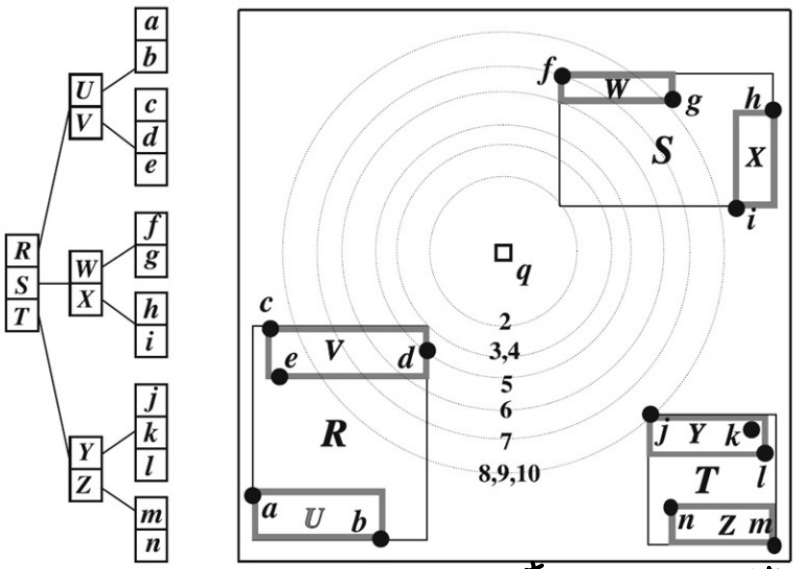
⇒⇒ I have to search all leaf nodes of the bounding boxes.

if they are overlapping

computationally expensive.

是否intersect? → 是 ok return
否 跳过，下一个

Continued R-tree



Step	Priority queue	Retrieved NN(s)
1	⟨S, R, T⟩	⟨⟩
2	⟨R, W, T, X⟩	⟨⟩
3	⟨V, W, T, X, U⟩	⟨⟩
4	⟨d, W, T, X, c, e, U⟩	⟨⟩

Step 5 finds d as the first NN (using Best First search)

2 – Using Indexes in SQL

Index Definition in SQL

- Some indexes are automatically created by the DBMS
  - For UNIQUE constraint, DBMS creates a non- clustered index.
    - For PRIMARY KEY, DBMS creates a clustered index
- You can create indexes on any relation (or view) – view: e.g., join result of 2 tables

Create an Index

create index <index-name> **on** <relation-name>
(<attribute-list>)

e.g.,

1. Create a clustered index on a table

   CREATE CLUSTERED INDEX index1 ON table1 (column1);

2. Create a non-clustered index with a unique constraints

   CREATE UNIQUE INDEX index1 ON table1 (column1 DESC, column2 ASC, column3 DESC);

   (A unique index is one in which no two rows are permitted to have the same index key value)

Drop an Index

drop index <index-name>

Note: Most database systems allow specification of type of index, and clustering

Specialized Index – filtered index

```
CREATE INDEX index1 ON table1 (column1)
WHERE Year > '2010';
```

(A filtered index is an optimized nonclustered index, suited for queries that select a small percentage of rows from a table. It uses a filter predicate to index a portion of the data in the table)

Specialized Index – spatial index

KD tree
R tree -> boundig box

```
CREATE SPATIAL INDEX index_name ON
table_name(Geometry_type_col_name) WITH ( BOUNDING_BOX = ( 0,
0, 500, 200 ) );
```


Things we need to know

- There is no point going through all index types for all data types
  - There is hundereds of them
  - Even each type would have many subtypes
    - E.g., MX-CIF quadtree is one quadtree type among many
  - Same with R-trees, etc
  - Same with many other index types

Things we need to know

- Given a data set, when uploading to the DBMS
  - Find the potential query types → frequency (经常到什么)
  - Research what indices that particular DBMS would have for that data type (数据类型)
  - Research for what queries you would better do on what index
  - Create index if you have large data
  - Monitor performance
  - Tune or create other indices
  - Your DMBS will have a version of the “create index” SQL statement

Quiz

1. Which one of the following statements is incorrect?

A. Hash index can be used for fast access to individual records with id type attribute.

B. Bitmap index is not commonly used in data analysis.

C. Quadtree index is suitable for nearest neighbour queries on multi- dimensional data.

D. A B+tree index is a balanced index

B+tree是balanced

!!

~~the length of the~~ paths from root node to all leaf nodes are the same~~

2. A restaurant database stores the name, cuisine type, GPS location, phone number as well as average expert rating of restaurants in a table. Customers can access the database via their web browsers and the database access website contains a map for querying as well. The main query type that is issued is a NN query by customers where given a point location on the map, the system shows the nearest restaurant. Which one of the following is the best index to create on this table for fast access?

A. R- tree index on GPS location

B. Bitmap index on cuisine type

C. Hash index on phone numbers

D. Hash index on GPS location

3. Which of the following statements is true for transaction processing?

A. Running transactions concurrently but not in isolation with each other is not desirable.

B. Transactions are expected to be run obeying Chemical properties such as properties of acids and metals.

C. Durability is the least important ACID property.

D. SQL queries are the main atomic units of execution in relational DBMSs

4. If I have a hash index given as follows, what should we expect to happen during query processing using this index for accessing the associated table?

The hash index is created on the "id" attribute of a table containing info about more than a billion social network users. Each "id" value is unique and generated per user at the time of creation of their social network account. This index is used to access the info associated with a user, i.e., given their id, e.g. by the system administrators.

For example, administrators access a user's info during a service phone call from the customer taking the id value and entering to the system to retrieve the associated customer row from the customer info table.

The DBMS uses a simple hash function which converts this big integer "id" number which has 20 decimal digits to binary form and takes the middle 4 bits out of the binary representation and uses it as the bucket number to access a bucket in the associated hash table.

In this case we expect that:

A. The hash function is simple but elegant and thus saves time in data access and cannot be better.

B. As this is a very simple hash function it will lead to slower than expected data access.

C. The hash index finds the data location on the disk as quickly as it could be as there is an even distribution of ids to hash table buckets.

D. None of the above is relevant as we do not know which four bits are read per "id" instance at this time, i.e., 0000 or 1111 or etc.