Database Management Systems (DBMS)

Lec 23: Query processing and optimization (Contd.)

Ramesh K. Jallu
IIIT Raichur

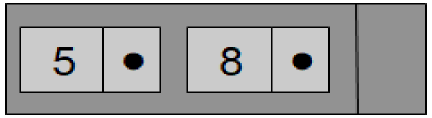
Date: 23/04/21

Today's plan

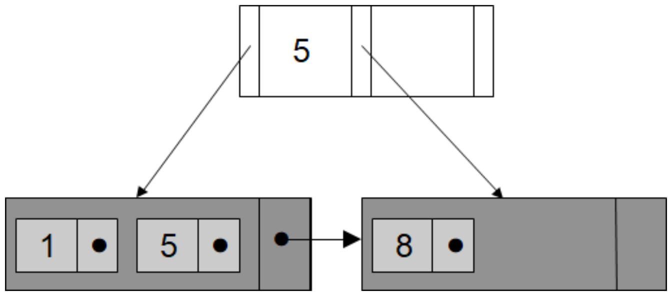
- Another example of B^+-tree insertion and deletion

- Other types of indexes
  - Hash indexing
  - Bitmap indexing

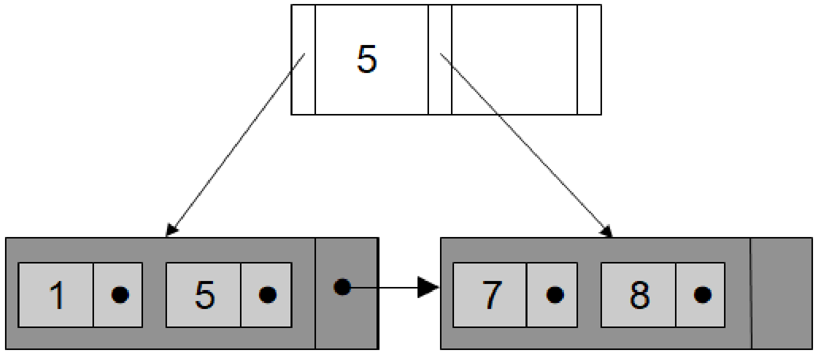
- Indexing on multiple keys



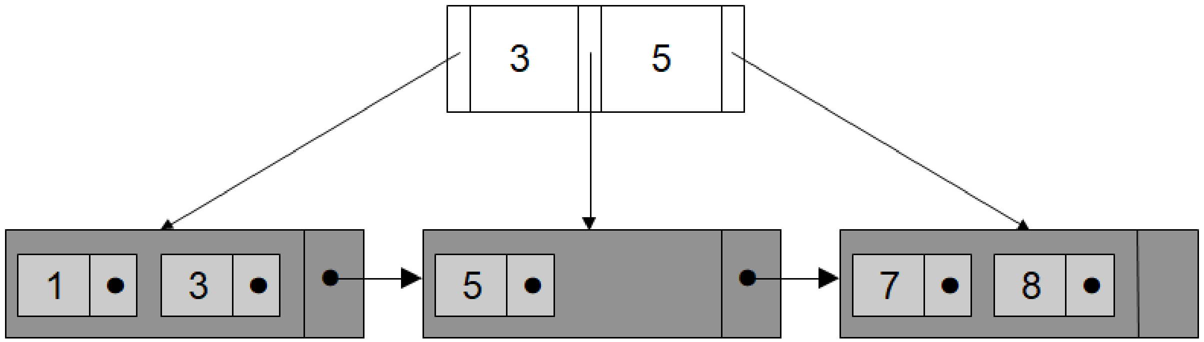
Insert 1



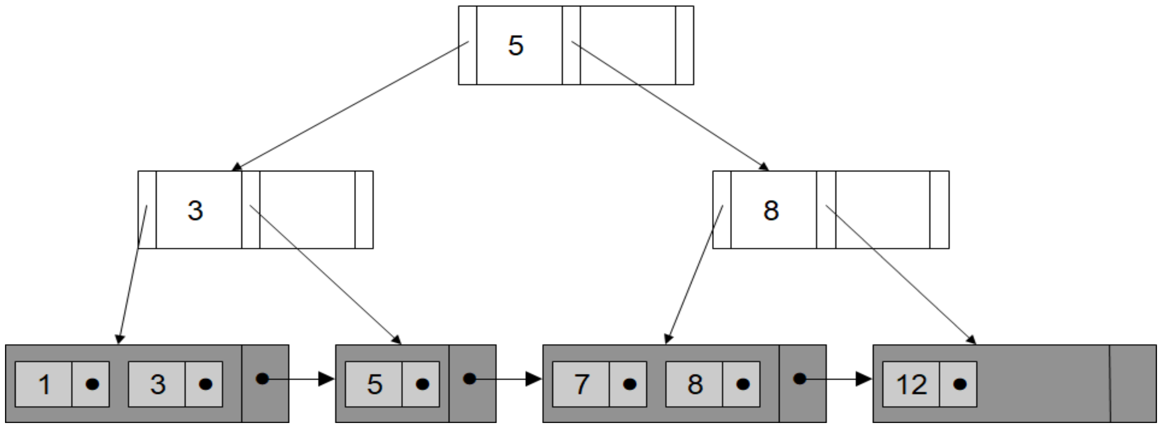
Insert 7



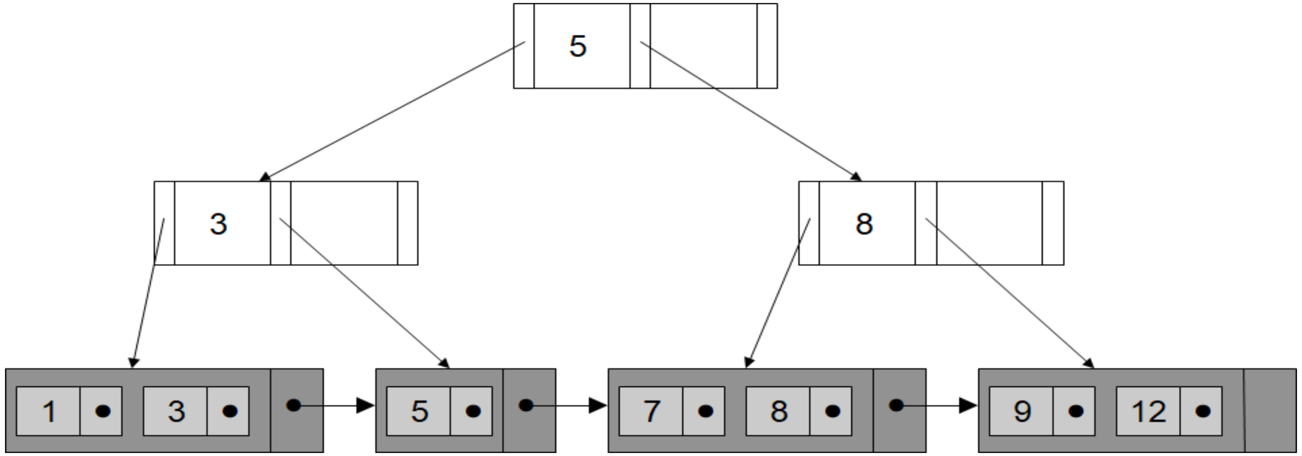
Insert 3



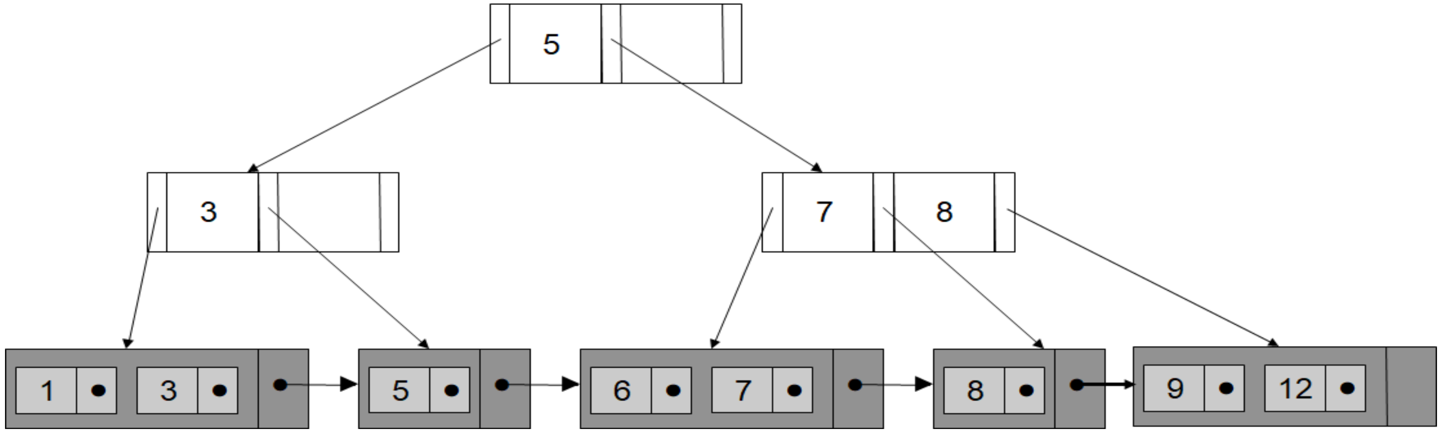
Insert 12

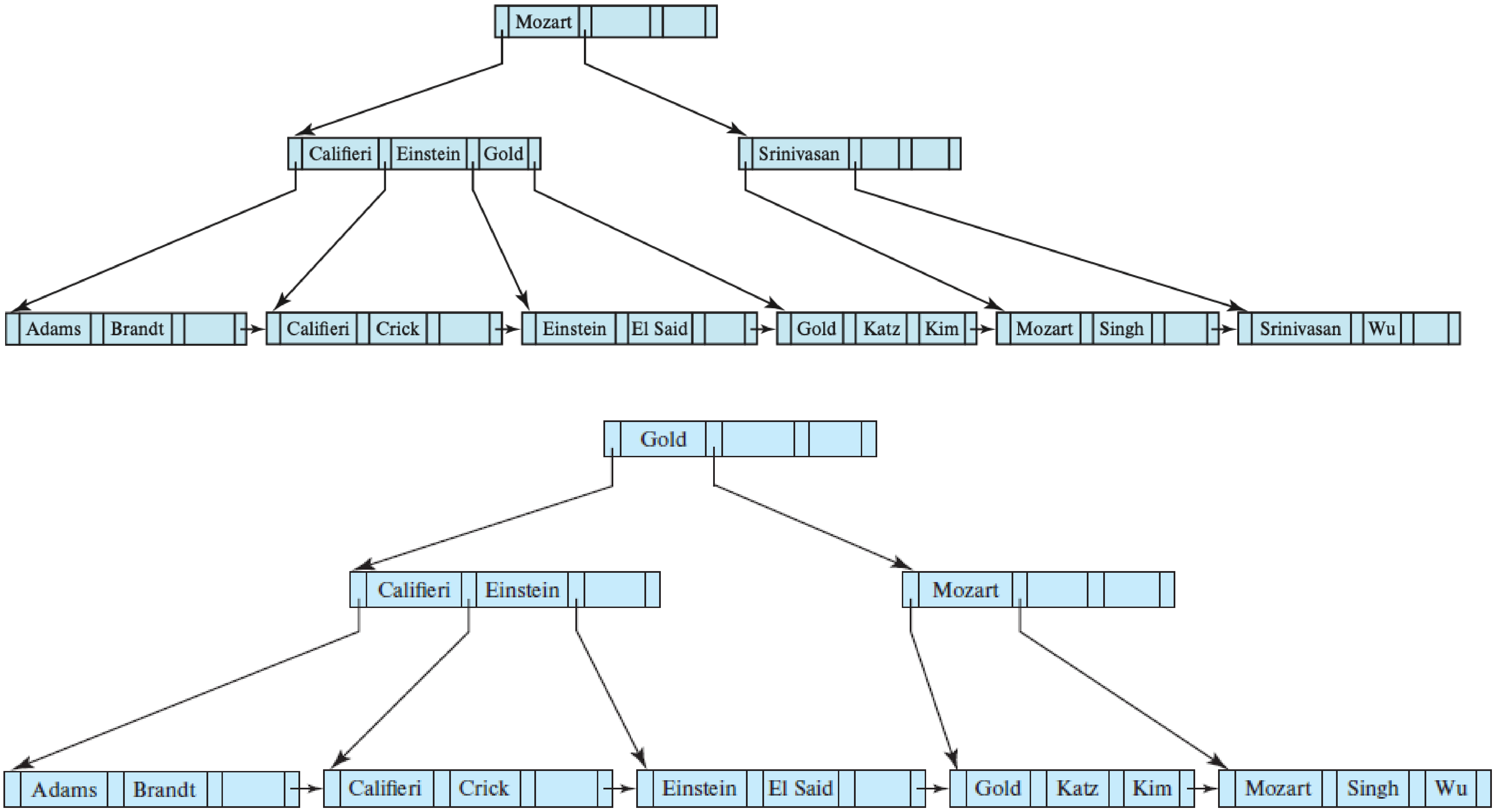


Insert 9

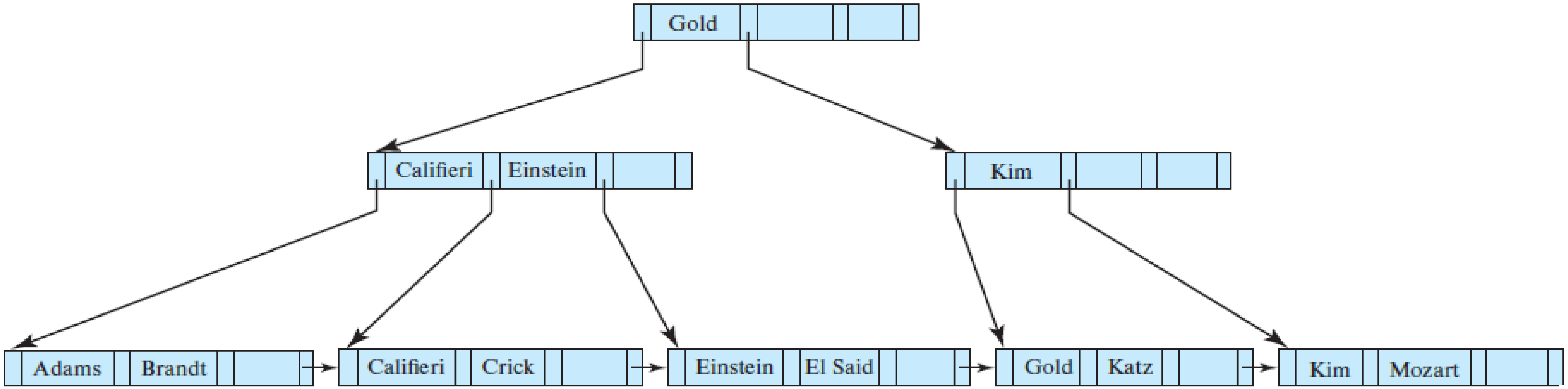


Insert 6

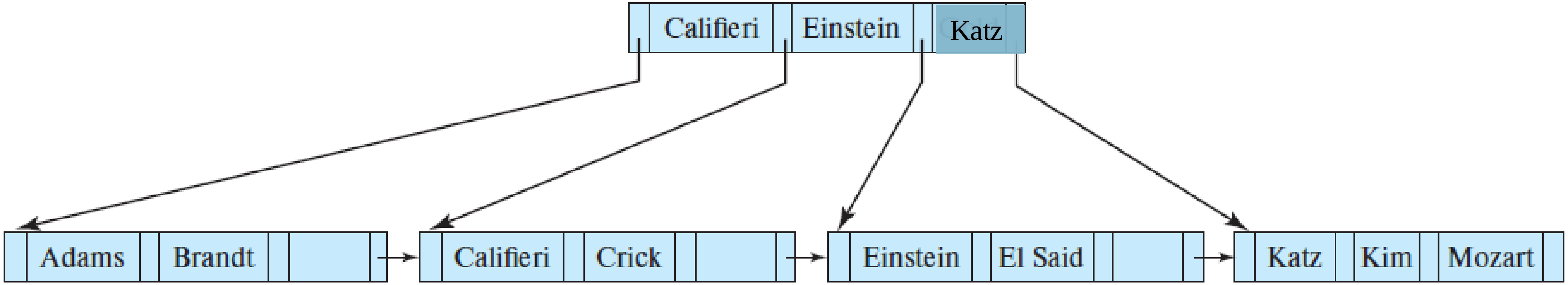




Deletion of “Srinivasan”



Deletion of “Singh” and “Wu”



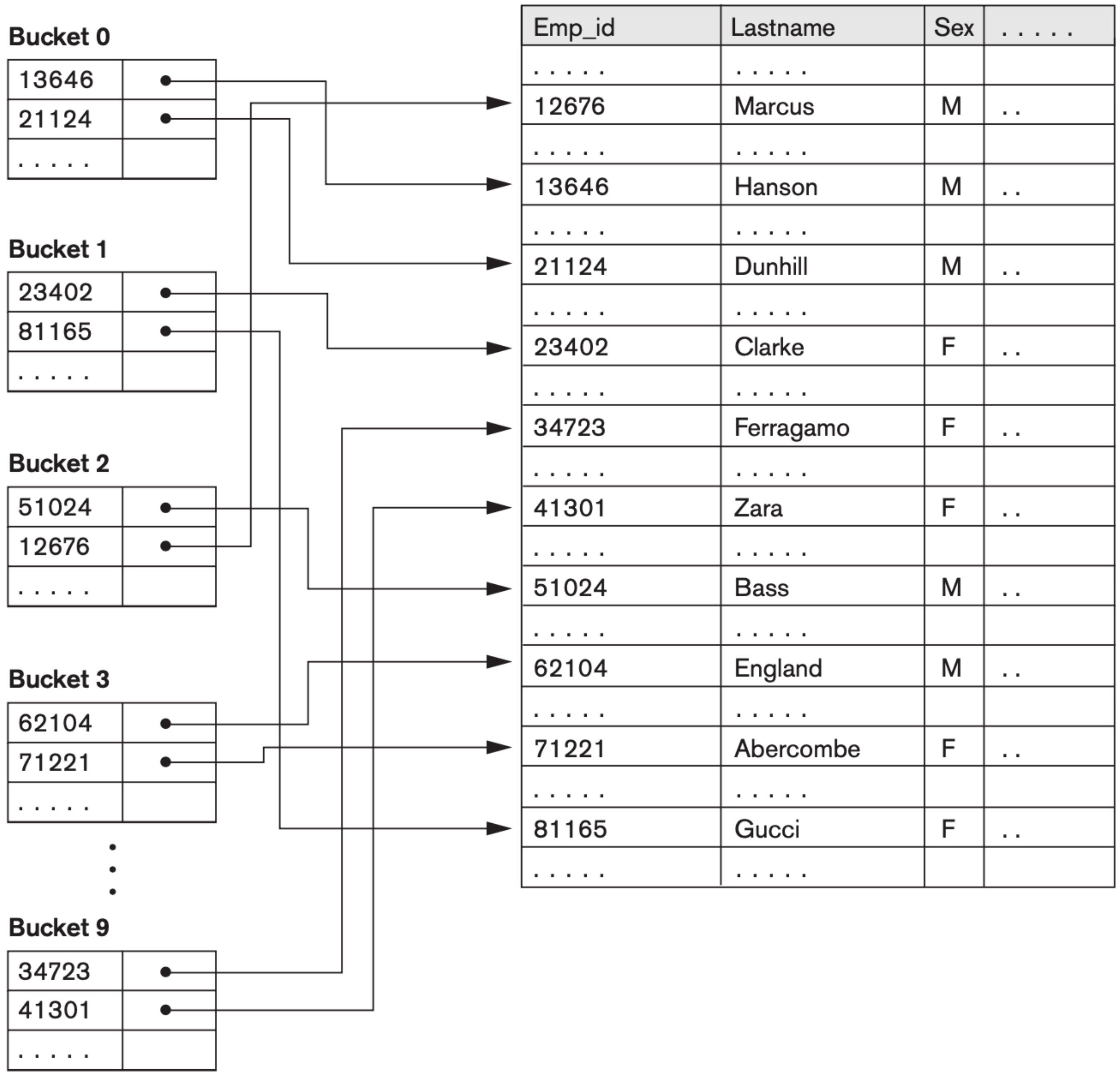
Deletion of “Gold”

Hash indexing

- A widely used technique for building indices
  - In-memory hash indexing
  - Disk-based hash indexing
- A ***bucket*** refers to a unit of storage that can store one or more records
- The index entries are of the type <*Key*, *Pr*>
- For in-memory hash indexing, a bucket could be a linked list of index entries or records
- For disk-based indices, a bucket would be a linked list of disk blocks

Hash indexing (Contd.)

- A ***hash function*** $h : K \rightarrow B$, where K is set of all search key values and B is set of all bucket addresses
- To insert a record with search key K_i, we compute $h(K_i)$, which gives the address of the bucket for that record and add the index entry for the record to the list at offset i
- Hash indices efficiently support equality queries on search keys
- Unlike B^+-tree indices, hash indices do not support range and inequality queries


Bucket 0
13646
21124
. . . . .
Bucket 1
23402
81165
. . . . .
Bucket 2
51024
12676
. . . . .
Bucket 3
62104
71221
. . . . .
Bucket 9
34723
41301
. . . . .
Emp_id
Lastname
Sex
. . . . .
12676 Marcus M
13646 Hanson M
21124 Dunhill M
23402 Clarke F
34723 Ferragamo F
41301 Zara F
51024 Bass M
62104 England M
71221 Abercombe F
81165 Gucci F

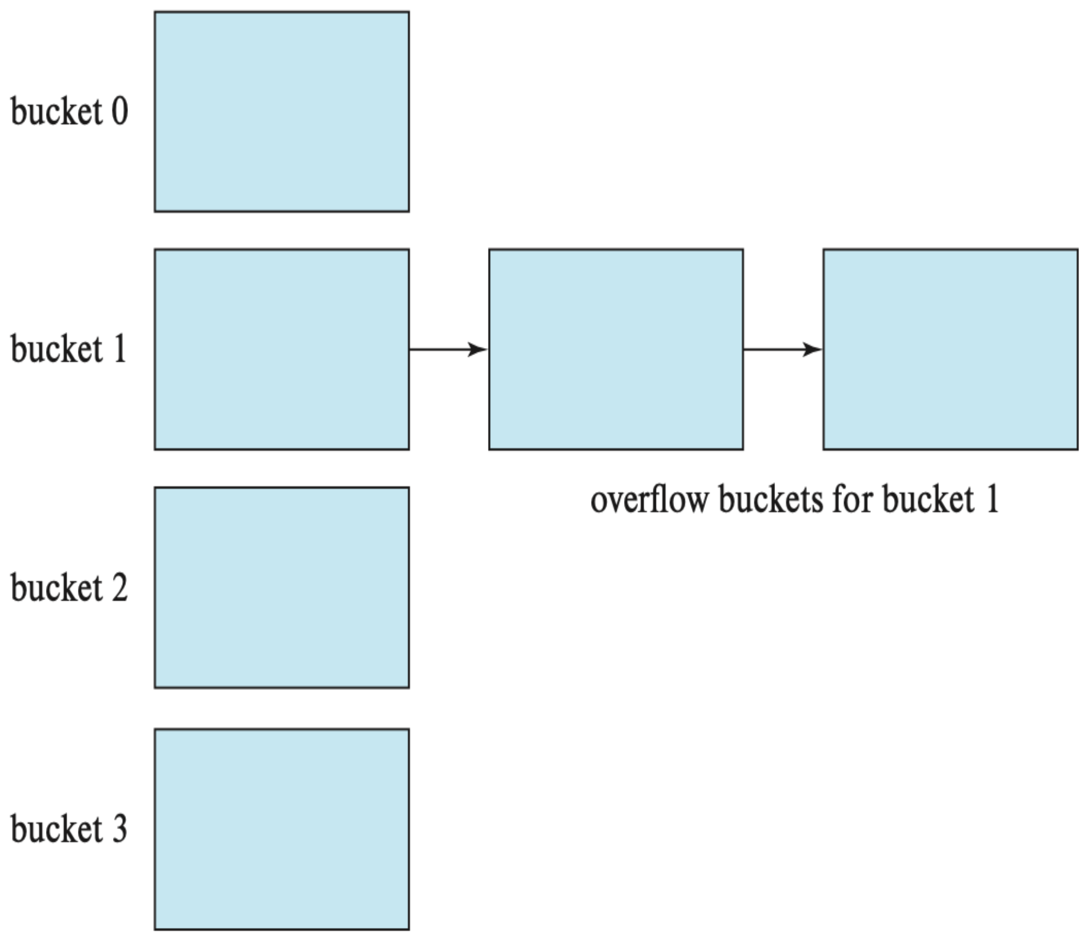
Hash indexing (Contd.)

- If the bucket does not have enough space or buckets are in insufficient number, a ***bucket overflow*** occurs

- We handle bucket overflow by using ***overflow buckets*** (aka ***external hashing***)

- If a record must be inserted into a bucket *b*, and *b* is already full, the system provides an overflow bucket for *b* and inserts the record into the overflow bucket, and this process continues if the overflow bucket if full

Hash indexing (Contd.)



- All the overflow buckets of a given bucket are chained together in a linked list

- Given search key *k*, we not only search bucket *h(k)*, but also the overflow buckets linked from bucket *h(k)*

- If the number of records that are indexed is known ahead of time, the required number of buckets can be allocated

Hash indexing (Contd.)

- Hash indexing where the number of buckets is fixed when the index is created is called ***static hashing***
    - We need to know how many records are going to be stored in the index
    - If over time a large number of records are added we need to search through a large number of records stored in a single bucket, or in one or more overflow buckets

- In ***dynamic hashing*** the hash index can be rebuilt with an increased number of buckets
    - Rebuilding the index has the drawback that it can take a long time if the relations are large

Hash indexing vs Normal indexing

1. The goal is to index and retrieve items in database as it is faster to search that specific item
2. It can't perform range or inequality search
3. With a good hash function it takes O(1)
4. The demand is less compare to B-tree/B^+-tree
5. Unsorted based on the index filed

1. The goal is to optimize or increase performance of database simply by minimizing number of disk accesses that are required when a query is processed
2. It can perform range and inequality search
3. Insertion and deletion take O(log n) time
4. B-tree/B^+-tree demand more space as every node contains several tree pointers and data/block pointers
5. Sorted based on the indexed filed

Indexes on multiple keys

- The indexes discussed so far, we have assumed that the primary or secondary keys on which files were accessed were single attributes
- To process many queries, retreival of data from multiple attributes is involved
- For example, *List the employees in department number 4 whose age is 59*
- If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes

Ordered Index on Multiple Attributes

- We will refer to keys containing multiple attributes as **compound keys**
- In ordered indexing an index is created on attributes $<A_1, A_2, \ldots , A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots , v_n>$
- A lexicographic ordering of these tuple values establishes an order on this composite search key
- For $u = <u_1,u_2, \ldots,u_n>$ and $v = <v_1,v_2, \ldots,v_n>$, we say $u \leq v$, if $u_1 < v_1$ or ($u_1 = v_1$ and $u_2 < v_2$) or . . . or ($u_1 = v_1$ and $u_2 = v_2$ and …. and $u_n = v_n$)
- An index on a composite key of n attributes works similarly to any index discussed in this chapter so far

Partitioned Hashing

- Partitioned hashing is an extension of static external hashing that allows access on multiple keys

- For a key consisting of *n* components, the hash function is designed to produce a result with *n* separate hash addresses

- The bucket address is a concatenation of these *n* addresses

- Search for the required composite search key is established by looking up the appropriate buckets that match the parts of the address in which we are interested

Example

- If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address
- Suppose that Dno = 4 has a hash address ‘100’ and Age = 59 has hash address ‘10101’
- To search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101
- Just to search for all employees with Age = 59, all buckets (8 of them) will be searched whose addresses are ‘000 10101’, ‘001 10101’, ... and so on

Grid files

- Another alternative is to organize a data file as a ***grid file***

- If we want to access a file on *k* keys we can construct a grid array with one linear scale (or dimension) for each of the search attributes

- The scales are made in a way as to achieve a uniform distribution of that attribute

- If the scales contain values $l_1, l_2, \ldots, l_k$ respectively for the k attributes, then the grid array file has a total of $l_1 \times l_2 \times \ldots \times l_k$ cells.

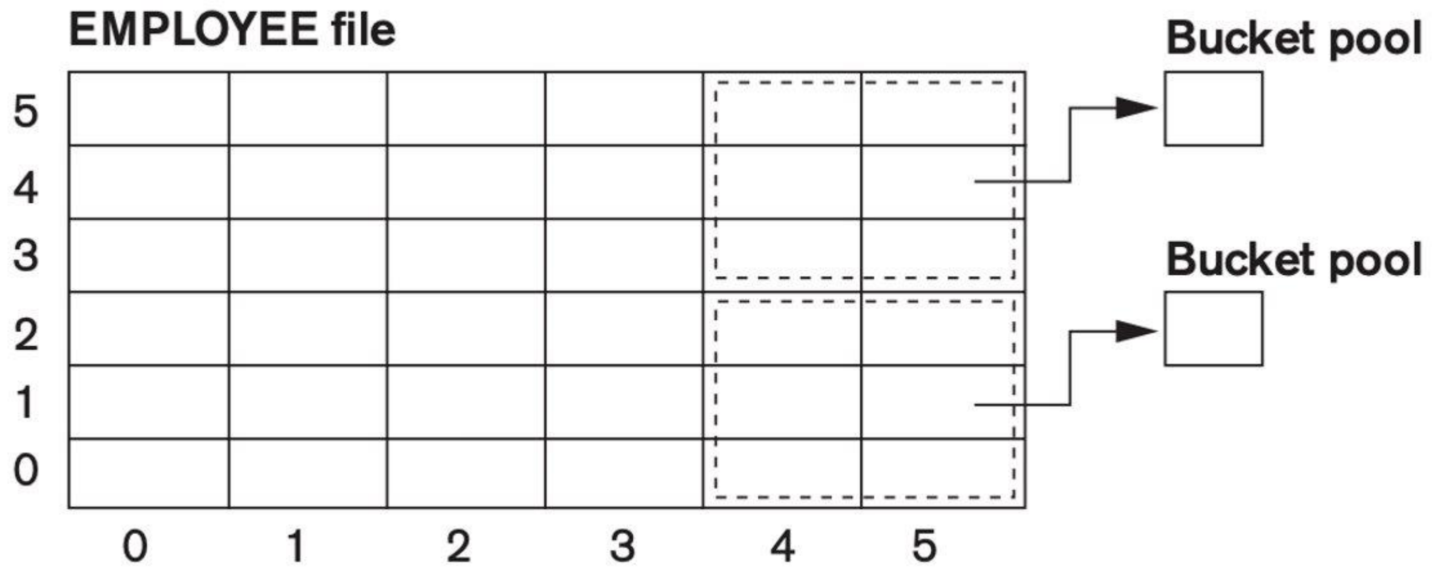
- Each cell points to some bucket address where the records corresponding to that cell are stored

Example

Dno

0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

Linear scale for Dno



Linear Scale for Age

0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

Grid files (Contd.)

- This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales

- If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells

- For example, a query for Dno $\leq$ 5 and Age > 40 refers to the data in the top bucket

Grid files (Contd.)

- **Basic idea**: The grid array allows a partitioning of the data file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions

- Grid files perform well in terms of reduction in time for multiple key access

- However, the disadvantage are
  1. The space overhead in terms of the grid array structure, and
  2. With dynamic data files, a frequent reorganization of the file adds to the maintenance cost

Bitmap indexing

- Bitmap indexing is used for relations that contain a *large number of rows*; columns that contain a *fairly small number of unique values*

- It creates an index for such columns, and each value in those columns is indexed

- To build a bitmap index on a set of records in a relation, the records must be numbered from *0* to *n* with an ***id*** that can be mapped to a block address/record address

- A bitmap index is built on one particular value of a particular column and is just an ***array of bits*** (corresponding to each unique value)

Bitmap indexing (Contd.)

- For a relation with n rows, bitmap index for a column C and a value V for that column contains n bits
- The i^{th} bit is set to 1 if the row i has the value V for column C; otherwise it is set to a 0
- If C contains the valueset $<v_1, v_2, \ldots, v_m>$ with m distinct values, then m bitmap indexes would be created for that column

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Bitmap indexing (Contd.)

- Queries can be answered based on bitwise operations
- For the query C_1 = *v1*, the corresponding bitmap for value v1 returns the Row_ids containing the rows that qualify
- For the query $C_1 = v_1$ and/or $C_2 = v_2$, the two corresponding bitmaps are retrieved and logical AND/OR operation is applied to yield the set of Row_ids that qualify
- Bitmap indexes are efficient in terms of the storage space that they need
- If we consider a file of 1 million rows with record size of 100 bytes per row, each bitmap index for any column would take 125 Kbytes

Bitmap indexing (Contd.)

- When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive
- Another bitmap, called the ***existence bitmap***, can be used to avoid this expense
- This bitmap has a 0 bit for the rows that have been deleted but are still physically present and a 1 bit for rows that actually exist
- Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index
- Inserted rows may be replaced with deleted rows to minimize the impact on the reorganization of the bitmaps

Function based indexing

- Function-based indexing has been introduced in the Oracle RDBMS
- Basic idea: Create an index for a computed value of an expression that involves one or more columns
- For example retreive the details of employees with (A + B)*(C-D) > 10000
- If we have a query that consists of expression and use this query many times, the database has to calculate the expression each time you execute the query

Summary

- Indexes should not be used on small tables
- Tables that have frequent, large batch updates or insert operations
- Indexes should not be used on columns that contain a high number of NULL values
- Columns that are frequently manipulated should not be indexed
- Maintenance on the index can become excessive

Thank you!