Université d'Ottawa | University of Ottawa

CSI2132 -Database I

Indexing & Hashing

Bisi Runsewe

Agenda

- Indexing - Basic Concepts
- Ordered Index
- B^+-Tree Index Files
- B- Tree Index Files
- Hashing
- Index on Multiple Keys
- Bitmap Index

Indexing

The Value of Indexing

- Scattering the records that represent tuples of a relation R among various blocks is not efficient
- Consider the query: **SELECT * FROM R.**
  - <u>Problem:</u>
    - Require examining every block in the storage system to find the tuples of R.
  - <u>Solution:</u>
    - Reserve some blocks, perhaps several whole cylinders, for R.
    - This will only enable scanning the tuples of R without scanning the entire data store.
    - Organization offers little help for a query like:

SELECT * FROM R WHERE a=10;

Indexing – Basic Concept

- Executing SQL queries takes a considerable amount of time to access data from the disk
- **Indexes:** Are access structures are needed to locate records in a relation directly
- Indexes are used by queries to find data from tables quickly
- Indexing optimizes the performance of the database by reducing the # of disks accesses to process queries
- Consider the query:

```
SELECT company_id, units, unit_cost
FROM Company
WHERE company_id = 18
```

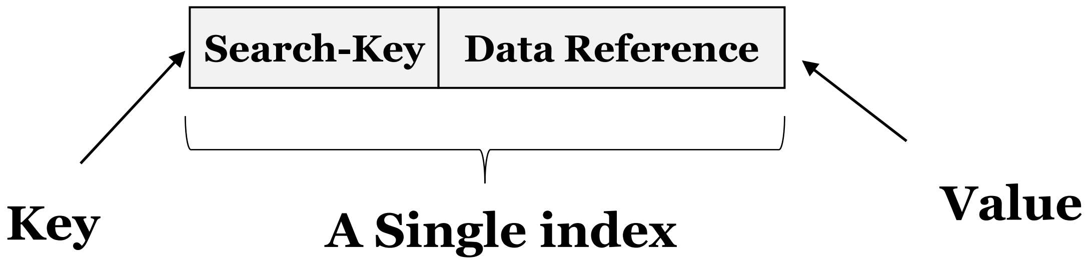
COMPANY_ID	UNIT	UNIT_COST
10	12	1.15
12	12	1.05
14	18	1.31
18	18	1.34
11	24	1.15
16	12	1.31
10	12	1.15
12	24	1.3
18	6	1.34
18	12	1.35
14	12	1.95
21	18	1.36
12	12	1.05
20	6	1.31

Indexing – Basic Concept

Ordered Record

COMPANY_ID	UNIT	UNIT_COST
10	12	1.15
10	12	1.15
11	24	1.15
11	24	1.15
12	12	1.05
12	24	1.3

- **An index:** Consists of a search key & a data reference
- **Indexes** – is a table containing only 2 columns using database keys
- 1st column – Search key that contains the primary or candidate key & stored in sorted order
- 2nd column – Data reference or pointer that holds the address of the disk block where the key value (tuple) can be found
- Index files are typically much smaller than the original file, they are also added as additional files of disks

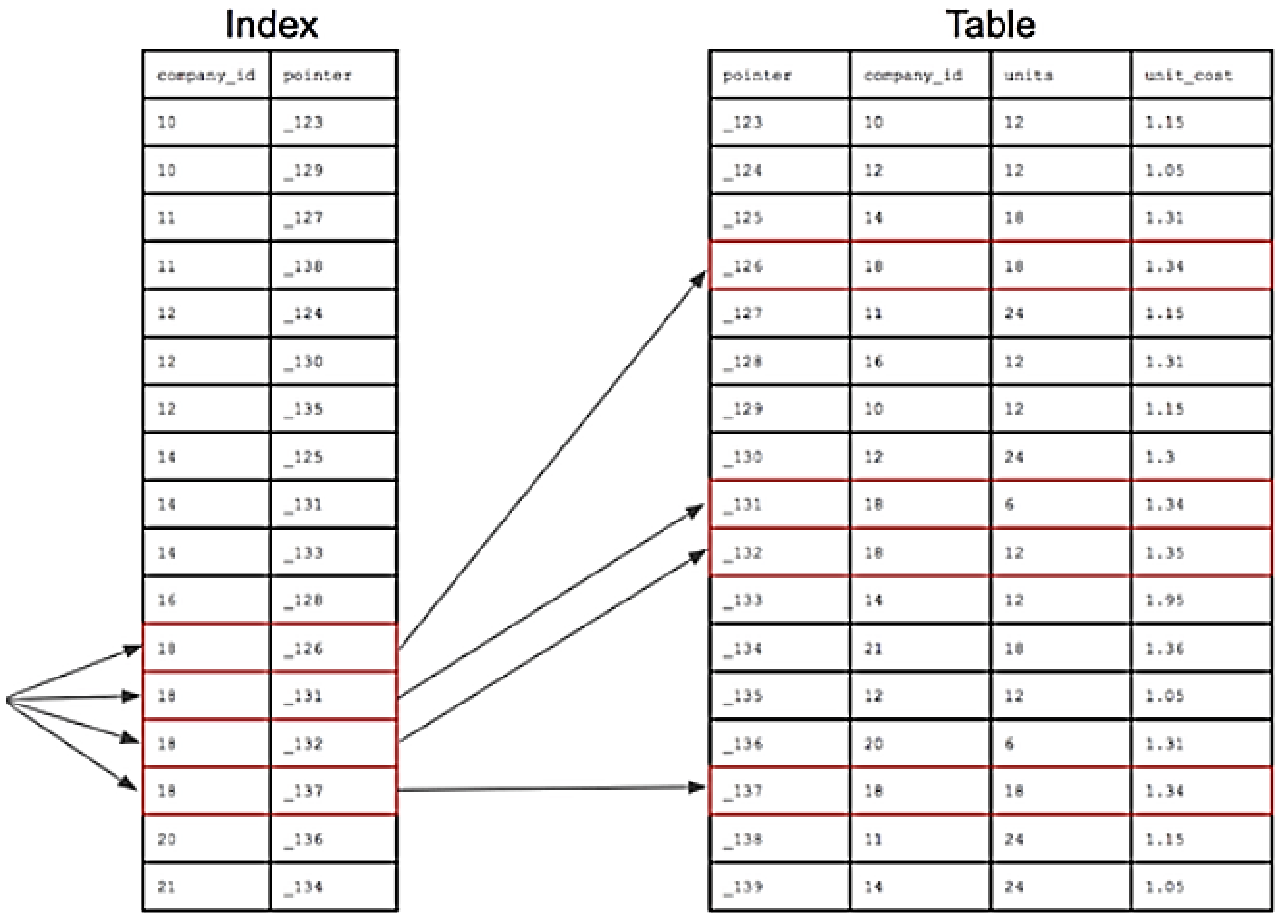


COMPANY_ID	POINTER
10	_123
10	_129
11	_127
11	_138
12	_124

Index file

Indexing – Basic Concept

User Query:
SELECT
company_id, units,
unit_cost
FROM Company
WHERE
company_id = 18



Index

company_id	pointer
10	_123
10	_129
11	_127
11	_138
12	_124
12	_130
12	_135
14	_125
14	_131
14	_133
16	_128
18	_126
18	_131
18	_132
18	_137
20	_136
21	_134

Table

pointer	company_id	units	unit_cost
_123	10	12	1.15
_124	12	12	1.05
_125	14	18	1.31
_126	18	18	1.34
_127	11	24	1.15
_128	16	12	1.31
_129	10	12	1.15
_130	12	24	1.3
_131	18	6	1.34
_132	18	12	1.35
_133	14	12	1.95
_134	21	18	1.36
_135	12	12	1.05
_136	20	6	1.31
_137	18	18	1.34
_138	11	24	1.15
_139	14	24	1.05

Index Evaluation Metrics

- Indexing techniques are evaluated based on the following factors:
- **Access Types:** Types of access that are supported efficiently
  - Includes finding:
    - Records with *a specified value* in the attribute
    - Records with an attribute value falling in *a specified range of values*
- **Access time**: Time it takes to find a particular data
- **Insertion time**: Time it takes to insert a data
- **Deletion time:** Time it takes to delete a data
- **Space overhead:** Additional space occupied by an index structure

Types of Indexes

- A variety of indexes are possible; each of them uses a particular data structure to speed up the search
- Two basic kinds of indexes:
  - **Ordered indexing:** Indices are sorted, making data searching faster
  - **Hash indexing:** Indices are based on the search keys being distributed uniformly across a range of “buckets” using a “hash function”
- Note:
  - No one technique is the best
  - Each technique is best suited for a particular database application.
  - There can be more than one index or hash function for a file

Ordered Indexes

Ordered Indexes

- Index entries are stored **sorted** on the search key
- A file may have several indices on different search keys

- **Clustering index:** An index whose search key *specifies the sequential order of the file*
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Non-clustering index**: An index whose search key *specifies an order different from the sequential order of the file*
  - Also called **secondary index**
- **Clustering on multiple keys**

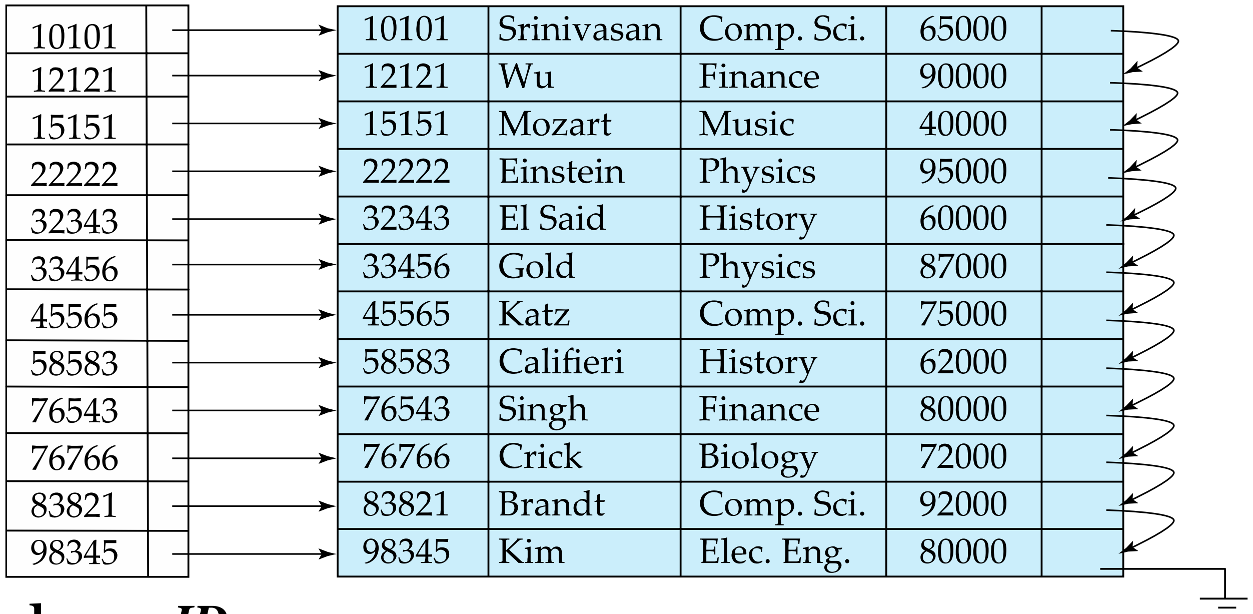
Clustering (Primary) Index

- A file can have at most one clustering (primary) index & several non-clustering (secondary) indexes

- <u>There are 2 types of Primary Index:</u>
  - **Dense index:** Has an index record *for every search key value* in the file.
  - **Sparse index:** Has an index record *for only some of the search key values* in the file
- Even with a sparse index, index size may still grow too large, leading to several disk reads
- <u>Solution:</u> Construct a sparse index on the index, i.e., **Multi-level Index**

Dense Index Files

- *Index record appears for every search-key value in the file*, e.g., index on *ID* attribute of *instructor* relation



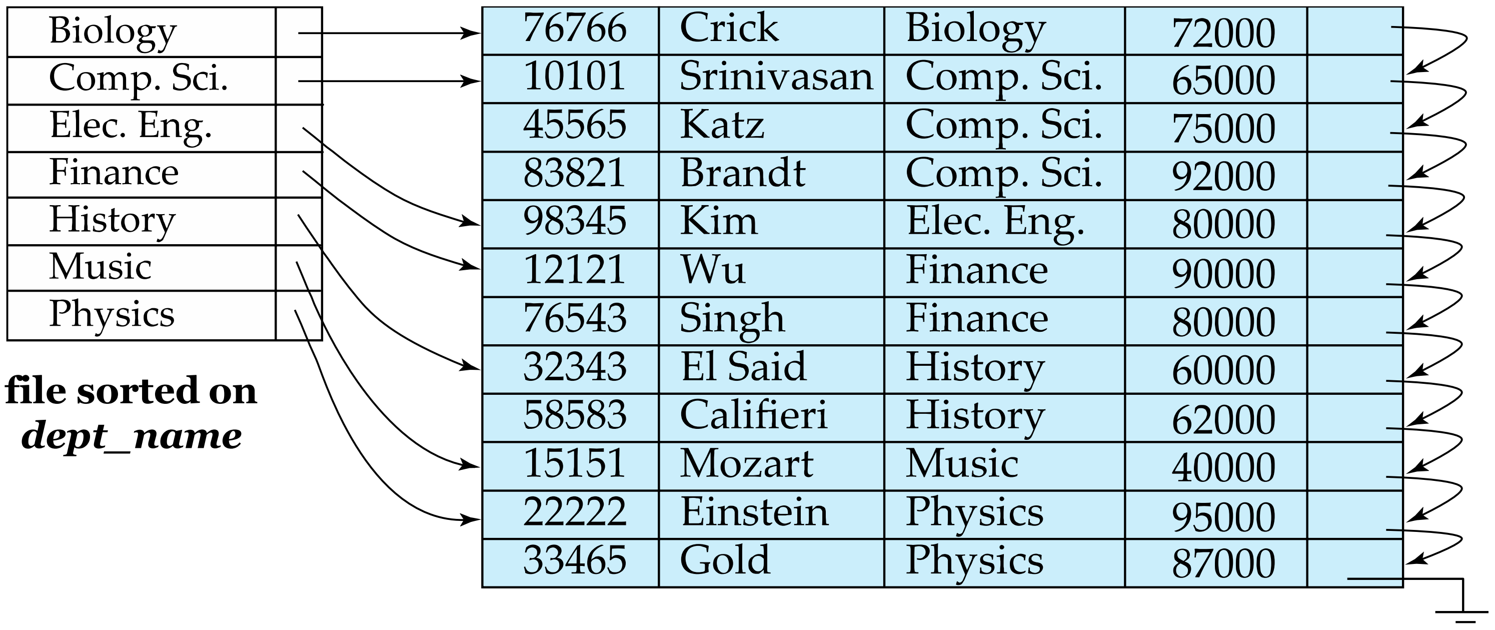
10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

index on *ID*

Dense Index Files

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



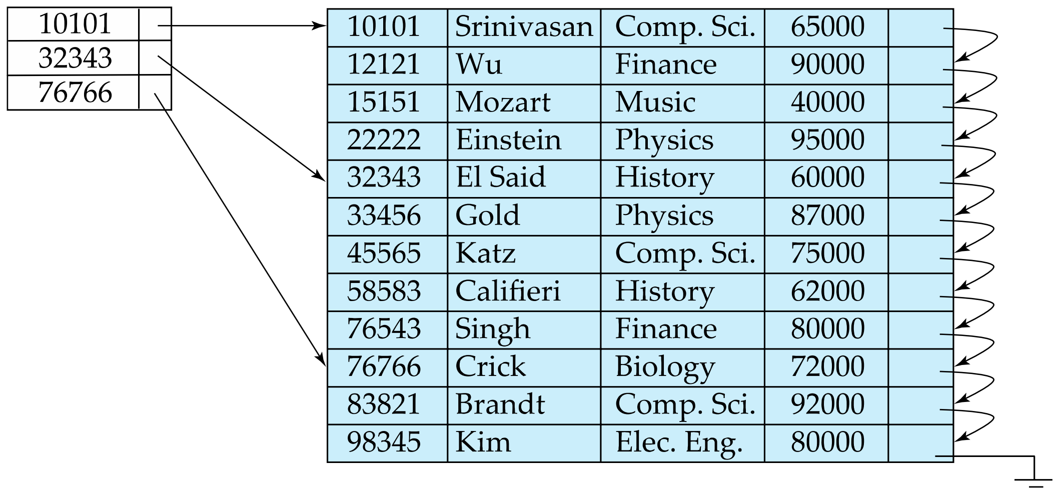
Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

file sorted on *dept_name*

Sparse Index Files

- Contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with **search-key value *K***:
  - Find index record with **largest search-key value < *K***
  - Search *file sequentially starting at the record* to which the index record points



10101
32343
76766

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

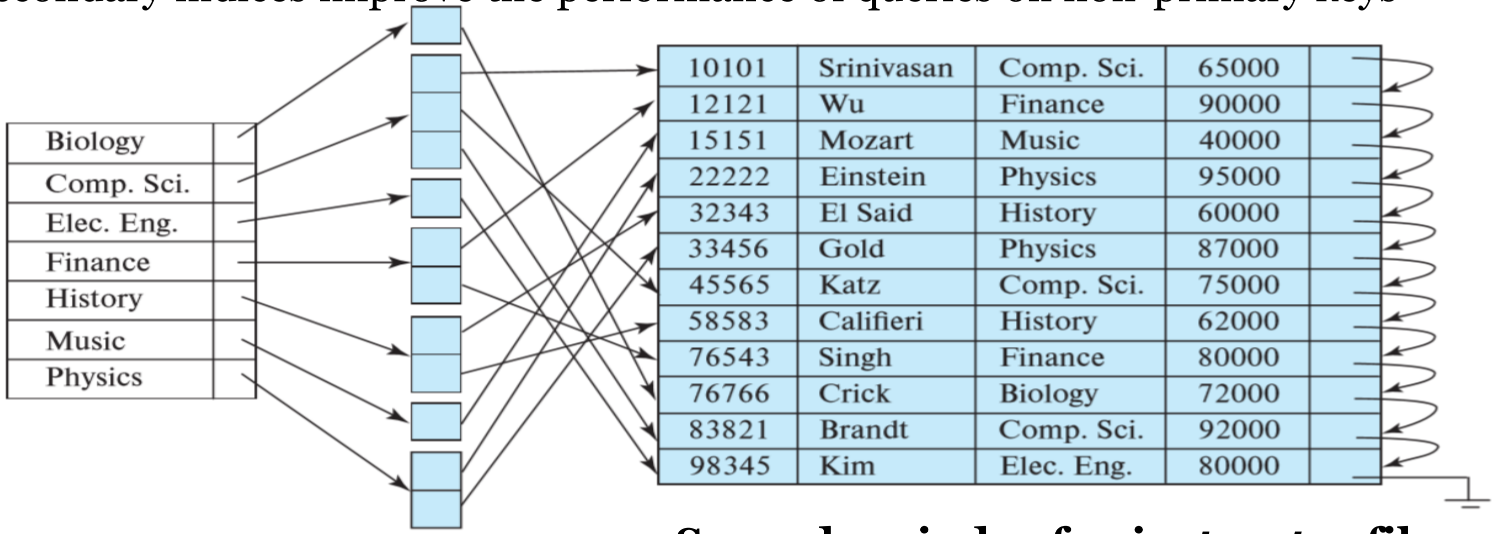
Sparse Index Files

- **Advantage over Dense Indexes:**
  - Requires ***less space & less maintenance overhead*** for insertions / deletions

- **Disadvantage:**
  - Generally **slower** than dense index for locating records.

Non-Clustering (Secondary) Index

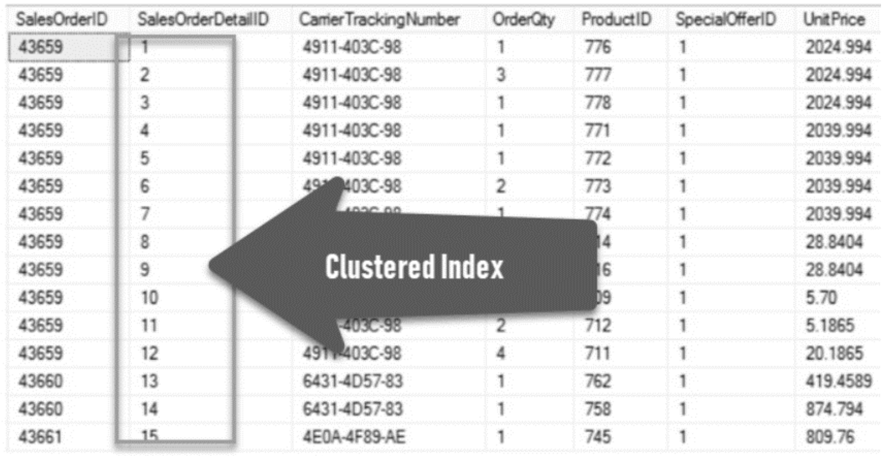
- **Secondary indices must be dense**, with an index entry for every search-key value & a pointer to every record in the file.
- Pointer does not point directly to the file but to a bucket that contains pointers to the file.
- Secondary indices improve the performance of queries on non-primary keys



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

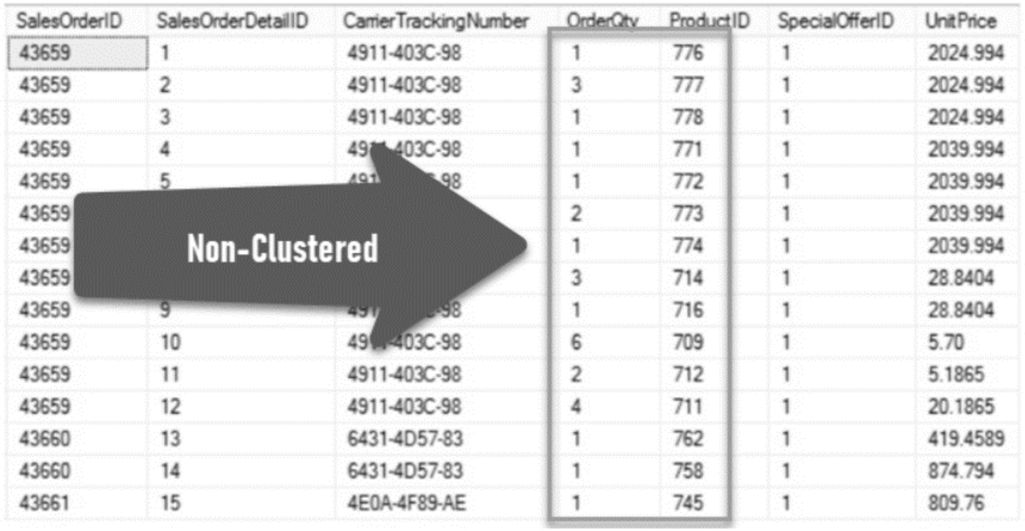
bucket

Secondary index for *instructor* file on noncandidate key *dept name*.



SalesOrderID	SalesOrderDetailID	CarrierTrackingNumber	OrderQty	ProductID	SpecialOfferID	UnitPrice
43659	1	4911-403C-98	1	776	1	2024.994
43659	2	4911-403C-98	3	777	1	2024.994
43659	3	4911-403C-98	1	778	1	2024.994
43659	4	4911-403C-98	1	771	1	2039.994
43659	5	4911-403C-98	1	772	1	2039.994
43659	6	49 403C-98	2	773	1	2039.994
43659	7	[illegible]	1	774	1	2039.994
43659	8	[illegible]	[illegible]	14	1	28.8404
43659	9	[illegible]	[illegible]	16	1	28.8404
43659	10	[illegible]	[illegible]	09	1	5.70
43659	11	-403C-98	2	712	1	5.1865
43659	12	491 403C-98	4	711	1	20.1865
43660	13	6431-4D57-83	1	762	1	419.4589
43660	14	6431-4D57-83	1	758	1	874.794
43661	15	4E0A-4F89-AE	1	745	1	809.76

Primary key



SalesOrderID	SalesOrderDetailID	CarrierTrackingNumber	OrderQty	ProductID	SpecialOfferID	UnitPrice
43659	1	4911-403C-98	1	776	1	2024.994
43659	2	4911-403C-98	3	777	1	2024.994
43659	3	4911-403C-98	1	778	1	2024.994
43659	4	49 403C-98	1	771	1	2039.994
43659	5	491 98	1	772	1	2039.994
43659	[illegible]	[illegible]	2	773	1	2039.994
43659	[illegible]	[illegible]	1	774	1	2039.994
43659	[illegible]	[illegible]	3	714	1	28.8404
43659	9	491 98	1	716	1	28.8404
43659	10	49 403C-98	6	709	1	5.70
43659	11	4911-403C-98	2	712	1	5.1865
43659	12	4911-403C-98	4	711	1	20.1865
43660	13	6431-4D57-83	1	762	1	419.4589
43660	14	6431-4D57-83	1	758	1	874.794
43661	15	4E0A-4F89-AE	1	745	1	809.76

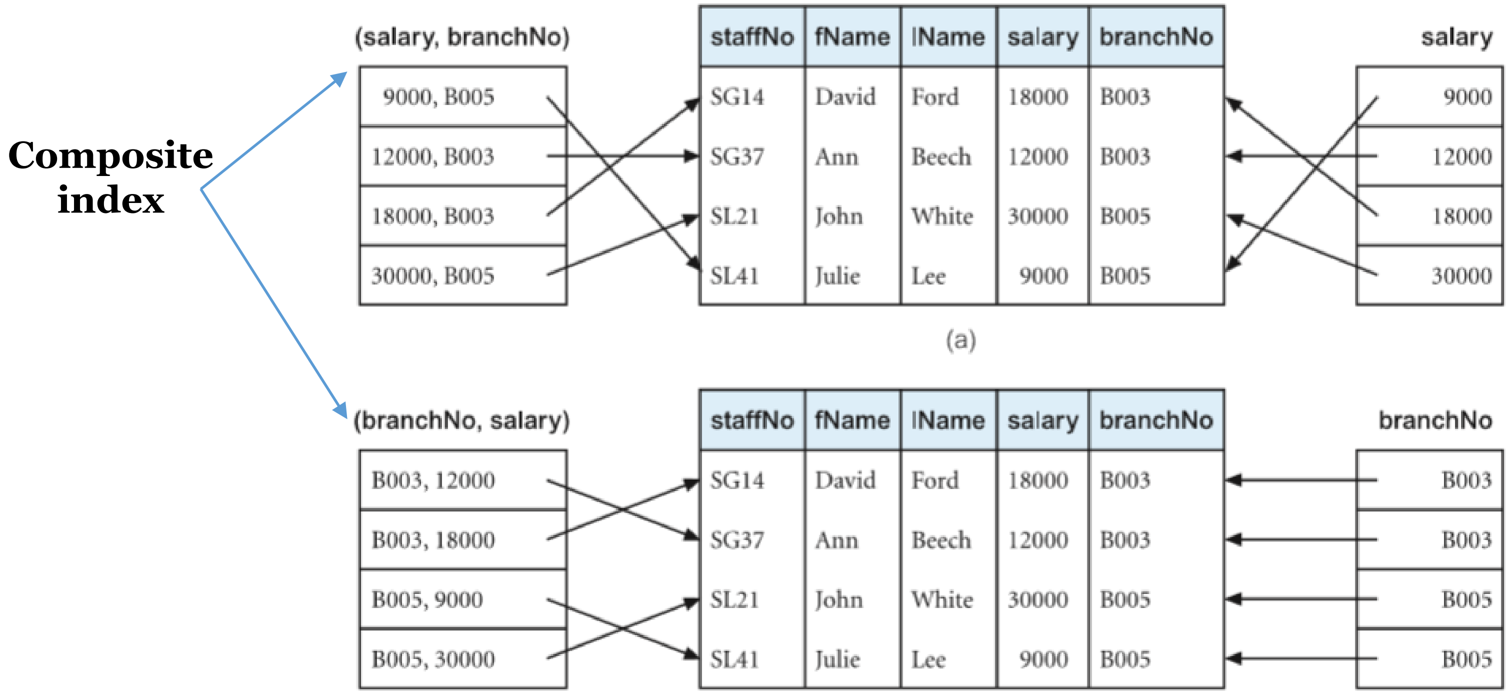
Secondary key

Indexes on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g., (John, 12121) < (John, 13514) and
      (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

Indexes on Multiple Keys

- 4 dense indexes on Staff table



(salary, branchNo)

9000, B005
12000, B003
18000, B003
30000, B005

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

salary

9000
12000
18000
30000

(a)

(branchNo, salary)

B003, 12000
B003, 18000
B005, 9000
B005, 30000

staffNo	fName	lName	salary	branchNo
SG14	David	Ford	18000	B003
SG37	Ann	Beech	12000	B003
SL21	John	White	30000	B005
SL41	Julie	Lee	9000	B005

branchNo

B003
B003
B005
B005

Multilevel Index

Multilevel Index

- When an index does not fit in memory, access can become expensive

- **Solution**:
  - Treat index kept on disk as another sequential file & construct a sparse index on it.
    - **Outer index:** A sparse index of the basic index
    - **Inner index**: The basic index file

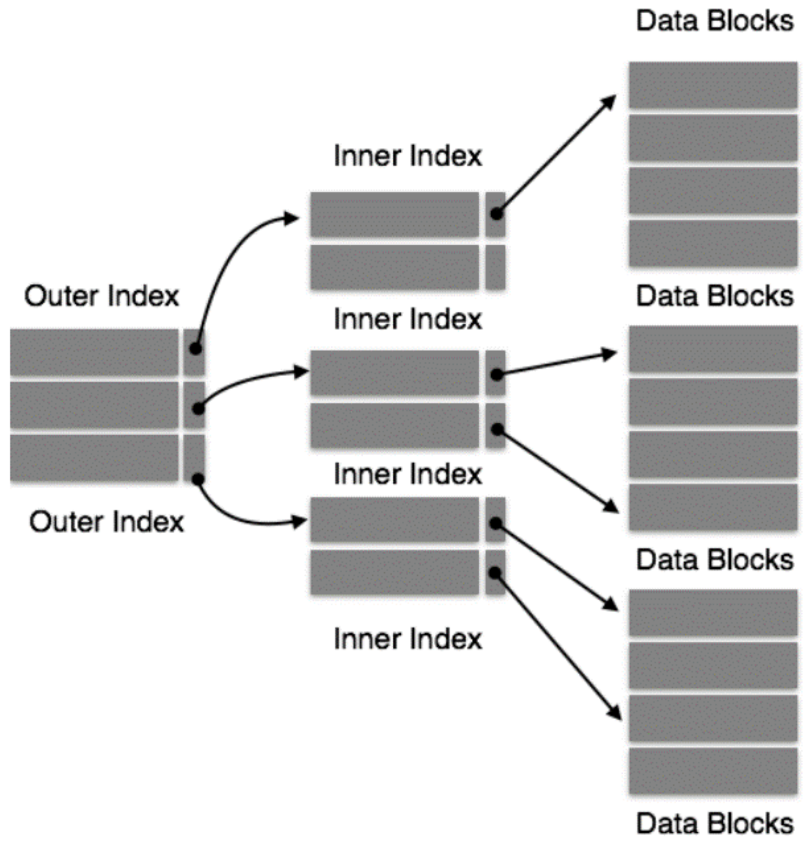
- Idea: whenever an outer index is too large to fit in main memory, another level of index can be created & so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index

- Regardless of what form of index is used, every index must be updated whenever a record is either ***inserted*** into or ***deleted*** from the file.

Issues:

- As indexes increase or decrease, this gives rise to the need for self-managed multilevel index.
- Also, performance is degraded
- No guidelines for the creation of multilevel index
- Solution: B+ trees



B^+ Trees Indexing

Tree-Structured Indexes

- Many DBMSs use a data structure called ***a tree*** to hold data or indexes.
- A tree consists of a ***hierarchy of nodes***.
- Each node in the tree (except the ***root*** node) has one ***parent*** node & zero or more ***child*** nodes.
- A root node has no ***parent***.
- A node that does not have any children is called a ***leaf*** node.
- The ***depth of a tree*** is the maximum number of levels between the root & leaf node in the tree
- The creation process is bottom-up
- Tree-structured indexes are ideal for range-searches & good for equality searches

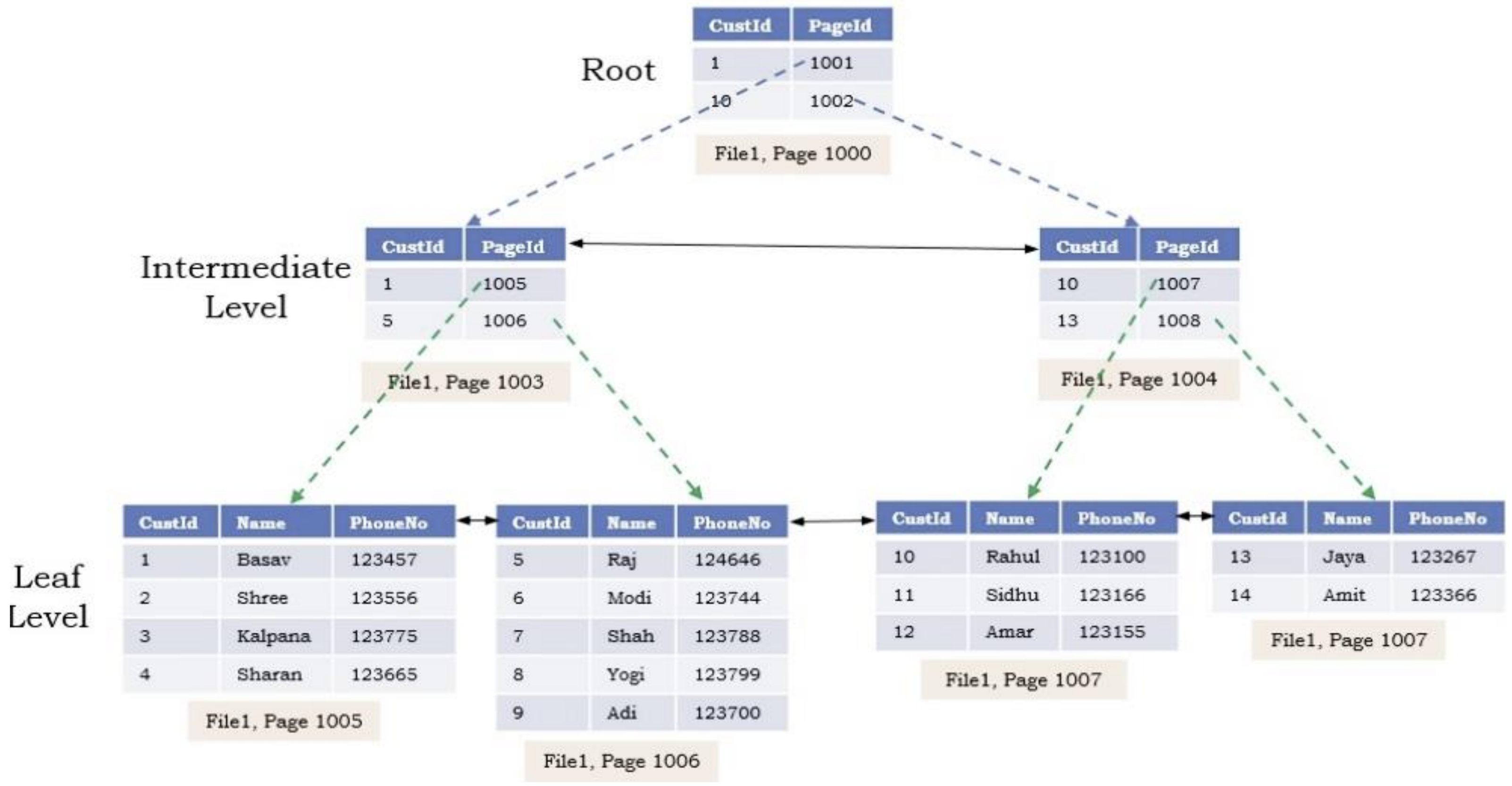
Tree-Structured Indexes

- Tree-structured indexes can be classified into:
  - **ISAM (I**ndexed Sequential Access Method): A static index structure that is effective when the file is not frequently updated
  - **B-tree:**
    - Very popular structure for organizing & maintaining large indexes
    - Provide guidelines for creating multilevel search trees
    - A B-tree index is a multilevel index but structured differently from multi-level index sequential files
  - **B+ tree:**
    - Variant of B-tree. Builds on trees

B^+ Tree Index

- B+ Tree is particularly efficient when data does not fit in memory & must be read from the disk
- Most queries can be executed more quickly if the values are stored in order
- B+ Trees enables storing row data in tree structure
- The "B" in B^+-tree stands for "balanced", i.e., all branches of the tree have same depth to ensure good performance for lookup, insertion & deletion
- **Consists of 3 layers:**
    - Root
    - Intermediate layer (any # of layers is possible)
    - Leaves

B+ Tree Index



CustId	PageId
1	1001
10	1002

CustId	PageId
1	1005
5	1006

CustId	PageId
10	1007
13	1008

CustId	Name	PhoneNo
1	Basav	123457
2	Shree	123556
3	Kalpana	123775
4	Sharan	123665

CustId	Name	PhoneNo
5	Raj	124646
6	Modi	123744
7	Shah	123788
8	Yogi	123799
9	Adi	123700

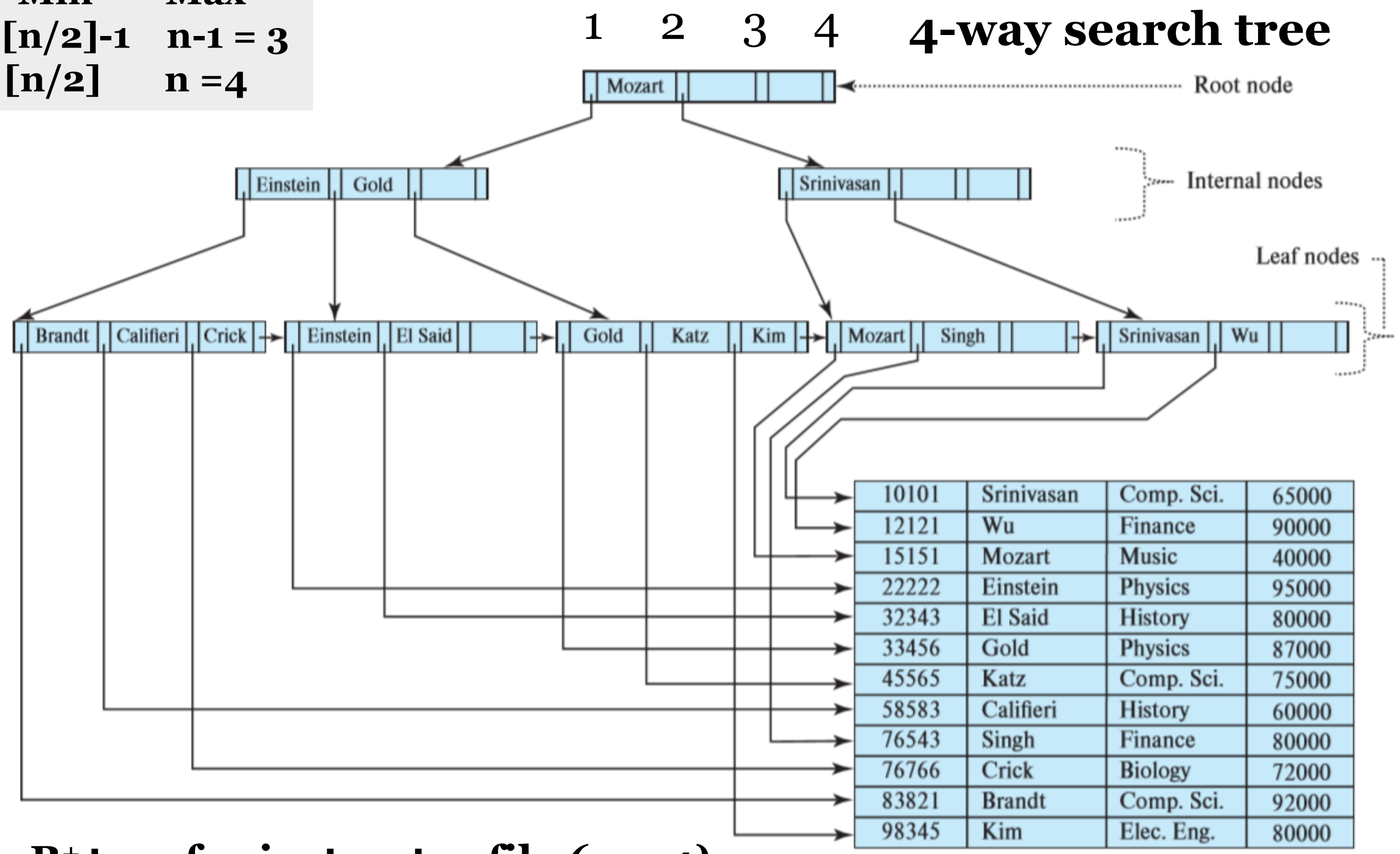
CustId	Name	PhoneNo
10	Rahul	123100
11	Sidhu	123166
12	Amar	123155

CustId	Name	PhoneNo
13	Jaya	123267
14	Amit	123366

- Leaf nodes contain actual row data
- Non-leaf nodes contain only pointers to other non-leaf nodes, or to leaf nodes.

B^+ Tree Index Example

	Min	Max
Key:	**[n/2]-1**	**n-1 = 3**
Pointer	**[n/2]**	**n =4**



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B^+ tree for instructor file *(n = 4)*

B+ Tree Node Structure

- Typical node:

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - $\boldsymbol{P_i}$ are **pointers to children or records (for leaf nodes)**
  - $\boldsymbol{K_i}$ are the **search key values**

  - The search-keys in a node are ordered

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

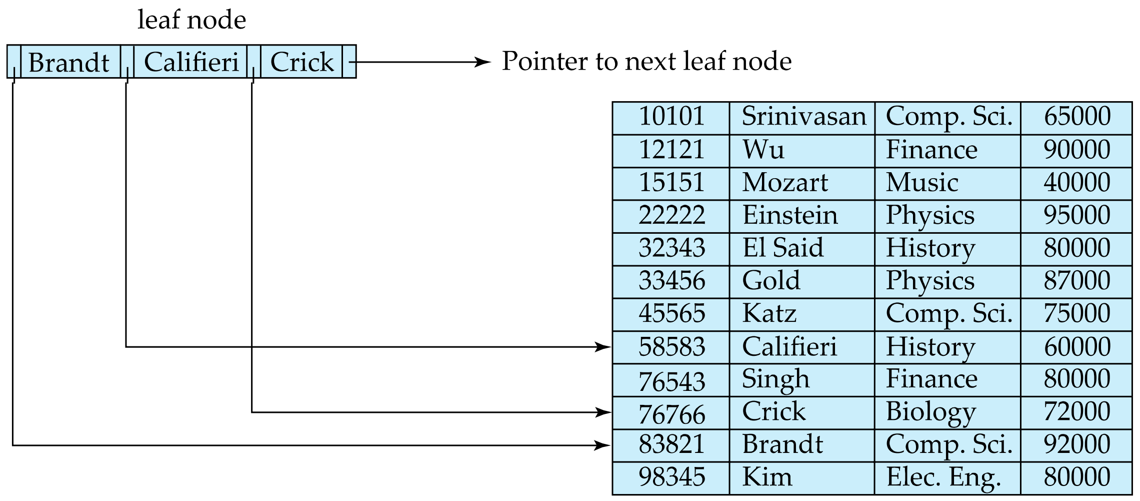
For a *n*-way search tree

- If search key value is >= key value, pointer to the left of key value is used to find the next node to be searched

Leaf & Non-leaf Nodes

- **Leaf node:** P_i points to a file record with search-key value K_i $(i = 1,..,n-1)$
  - If L_i, L_j are leaf nodes, i < j, then every search-key in L_i <= L_j' s search-key values.

One leaf node of a B+-tree for *instructor* file, where $n = 4$ & search key is *name*.



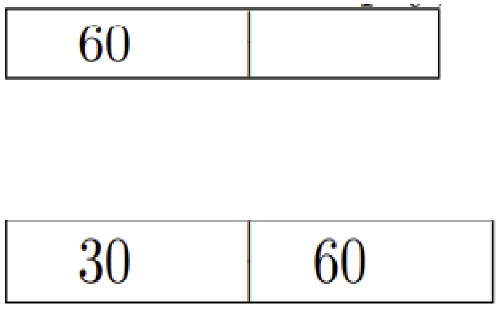
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- **Non-leaf node:** A form a *multi-level sparse* index on the leaf nodes.

B+ Tree Construction

- Sequence of keys: 60, 30, 90…
- If a node is empty, then the data are added on the left.
- If a node has one entry, then the left takes the smallest valued key & the right takes the biggest.
- If no sibling to the left, check the right
- If no sibling to the right, split the node
- If odd number of elements, put more on in the right leaf node
- Values are ordered from left to right

Count > Min or Max



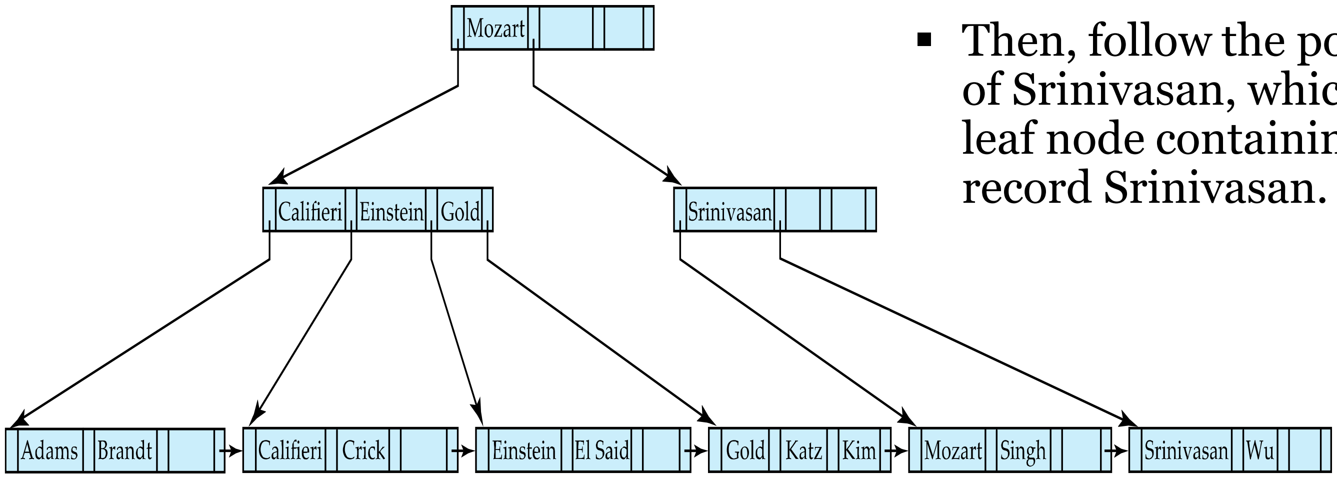
Queries on B^+ Trees

Lookups

Queries of B+ Trees

- B+-trees can be used to find all records with search key values in a specified range [*lb*, *ub*]
- These queries are called **Range Queries.**

- To locate Srinivasan, start from the root node.
- If Srinivasan, is greater than Mozart, follow the pointer to the right, which leads to the second level node containing the key values Srinivasan.
- Then, follow the pointer to the left of Srinivasan, which leads to the leaf node containing the address of record Srinivasan.



Updates on B$^+$ Trees

Insertion & Deletion

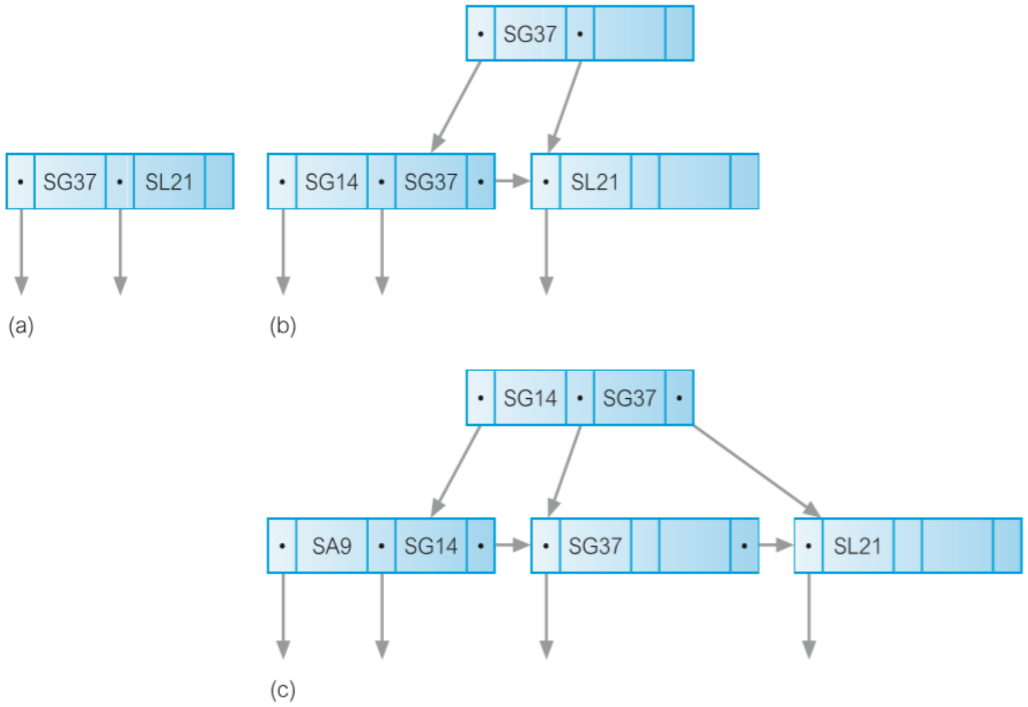
Updates on B+ Trees

- More complicated than queries (lookups), as they may require splitting or combining nodes to keep the tree balanced

<u>When splitting or combining are not required</u>

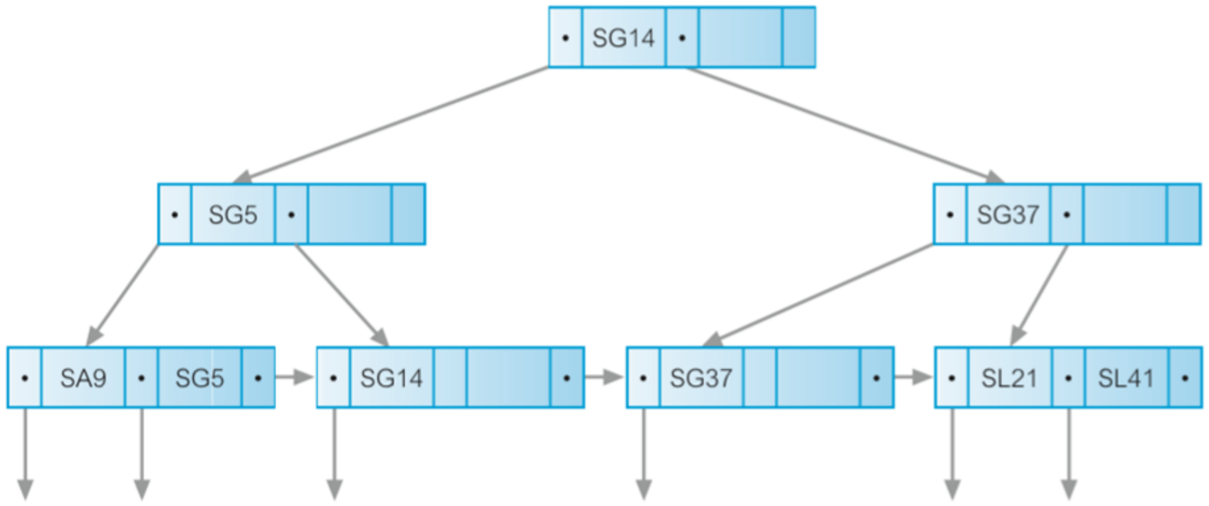
- **Insertion works as follows:**
  - Find leaf node where search key value should appear.
  - If value is present, add new record to the bucket of pointers.
  - If value is not present, insert value in leaf node based on order
  - Create a new bucket & insert the new record.
- **Deletion works as follows:**
  - Find record to be deleted & remove it from the bucket.
  - If bucket is now empty, remove search key value from leaf node.

Example: B+ Tree Insertion



Using a 3-way search tree

- (a) shows the construction of a tree after the insertion of the first two records SL21 & SG37.
- (b) The **next record to be inserted is SG14.** The node is full, so we must split the node by moving SL21 to a new node.
- Also, we create a parent node consisting of the rightmost key value of the left node.
- (c) The **next record to be inserted is SA9**



- SA9 should be located to the left of SG14, but again the node is full.
- We split the node by moving SG37 to a new node.
- We also move SG14 to the parent node.
- The **next record to be inserted is SG5.** SG5 should be located right of SA9 but again the node is full.

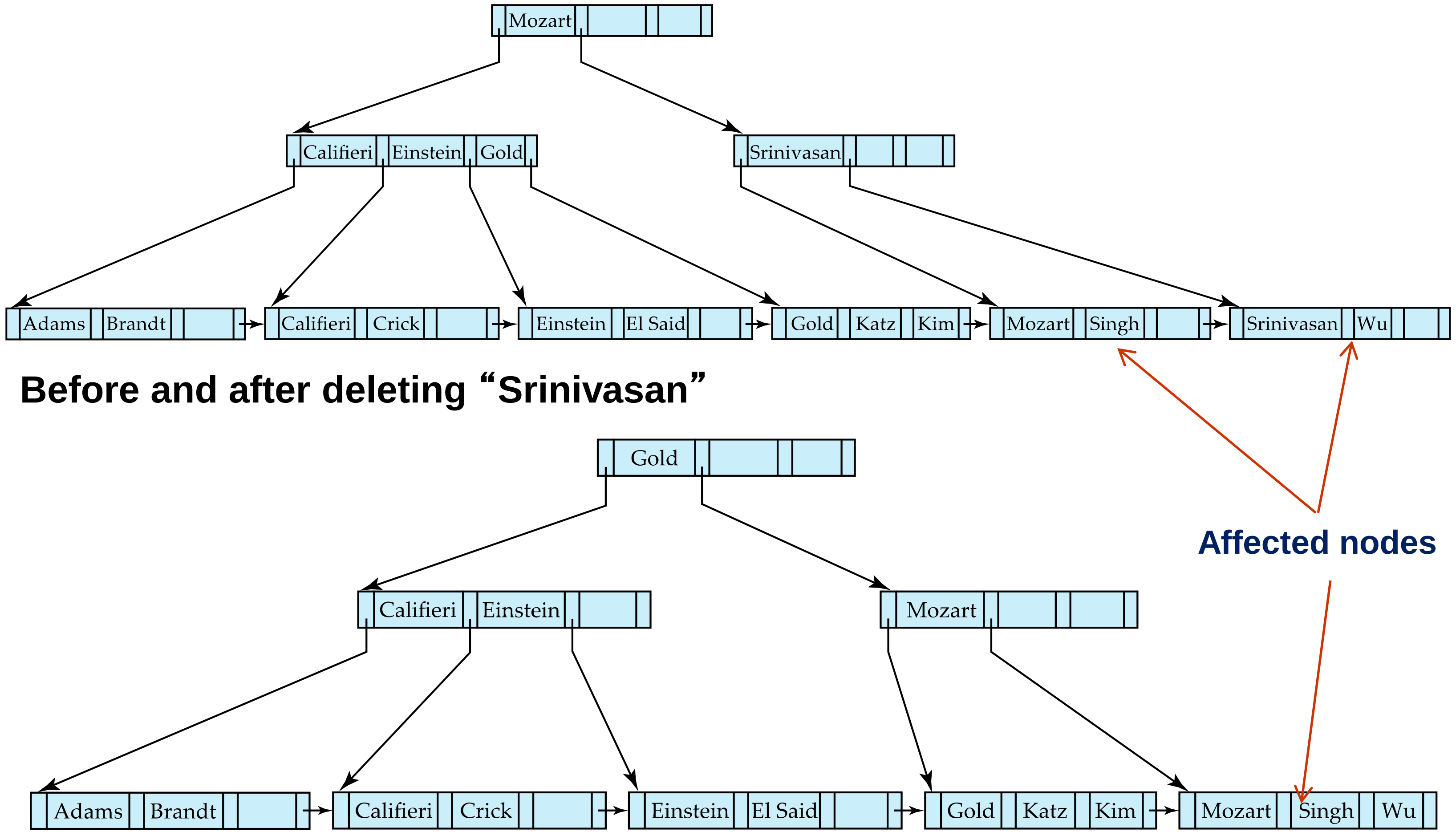
- We split the node by moving SG14 to a new node & add SG5 to parent node.
- However, the parent node is also full & has to be split.
- A new parent node has to be created again.
- Finally, **record SL41** is added to the right of SL21.

B+ Tree Deletion

B+ tree entries are deleted at the leaf nodes.

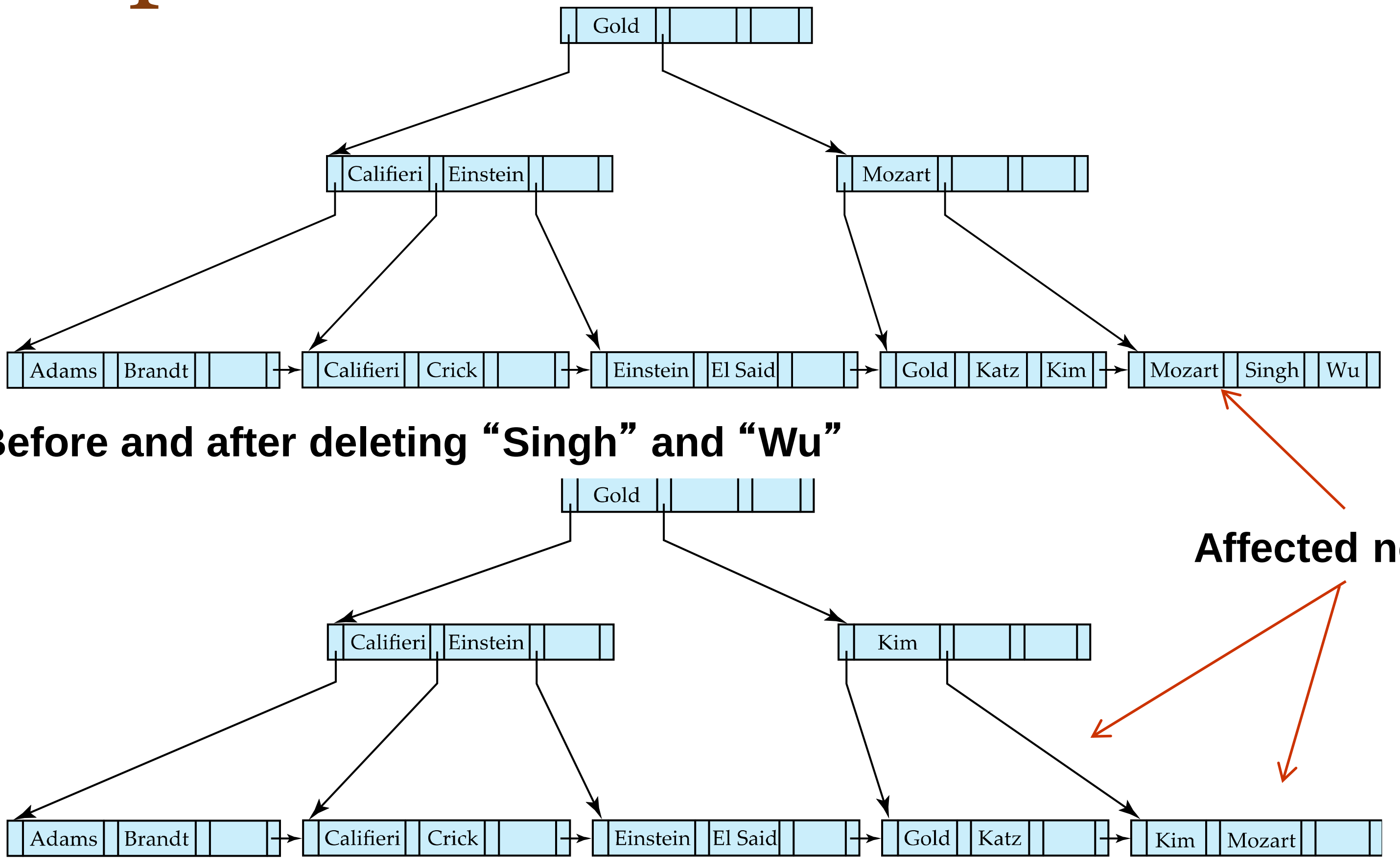
- Target entry is searched & deleted.
  - If it is an internal node, delete & replace with the entry from left position.
- After deletion, underflow is tested,
  - If underflow occurs, distribute entries from nodes left to it.
- If distribution is not possible from left, then
  - Distribute from nodes right to it.
- If distribution is not possible from left or from right, then
  - Merge node with left & right to it.

Example: B+ Tree Deletion



Deleting "Srinivasan" causes merging of under-full leaves

Example: B+ Tree Deletion



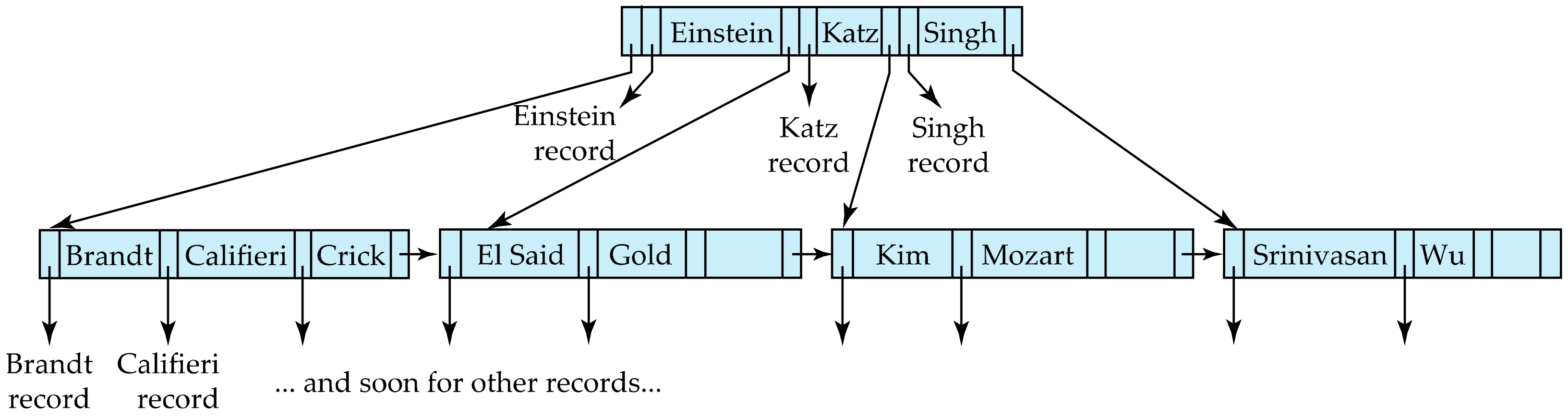
- Leaf containing Singh & Wu became underfull & **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

Next Class..

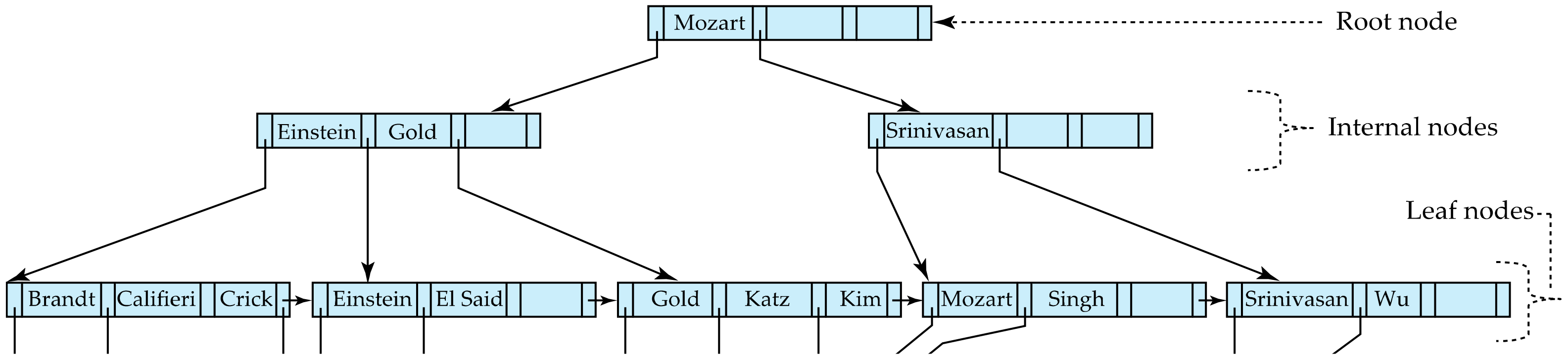
Indexing Strings

- Creating B+ tree index on string-valued attributes **raises 2 problems:**
  1. **Variable length strings as keys:** Different nodes can have different fanouts (# of pointers or children per node)
  2. **Strings can be long, leading to a low fanout** & a correspondingly increased tree height.
- It is important to increase fanout as this allows to direct searches to the leaf level more efficiently
- **Solution:**
  - **Prefix Compression:** Store a prefix of each search key value sufficient to distinguish between the key values in the subtrees that it separates
    - E.g., "Silas" and "Silberschatz" can be separated by "Silb"

Example: B-Tree Index File



B-tree (above) and B+-tree (below) on same data



Hashing

Hashing

- A hash table is an effective data structure for fast retrieval of data no matter how much data there is
- Hashing is widely used in database indexing, caching, error-checking, etc.
- Consider a simple 1D array variable:

Einstein	Katz	Singh	Gold	Crick	Kim	Wu	Brandt	Mozart
0	1	2	3	4	5	6	7	8

- To find an item on the array, a brute force approach can be used, e.g., a linear search which involves checking each item
- If the index # is known, the value can be retrieved quickly
- Question: How can we know which element of the array contains the value? Each index # can be calculated using the value itself
- .

Hash Function

- Hashing uses hash functions with search keys as parameters to generate the address of a data record
- **Hash Function:** algorithm is applied to a search key to transform it to a relatively small # that corresponds to a position in the hash table
- i.e., a function that maps values in a search field into a range of buckets
- **Bucket:** denotes a unit of storage that can store one or more records, typically a disk block, can be larger or smaller
- # of buckets = # of search key values stored in the database
- <u>Qualities of a Hash function</u>
    1. The worst hash function maps all keys to the same bucket
    2. The best hash function maps all keys to distinct addresses

Hash File Organization

- Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure
- In **hash file organization**, instead of record pointers for hash index, buckets store the actual records using a **hash function**

<u>Methods for generating hash functions:</u>

- For numeric keys, divide the *key* by the # of addresses, *n*, and take the remainder

$$Address = Key \bmod n$$

- For alphanumeric keys, divide the sum of the ASCII codes in a key by the # of available addresses, n, and take the remainder

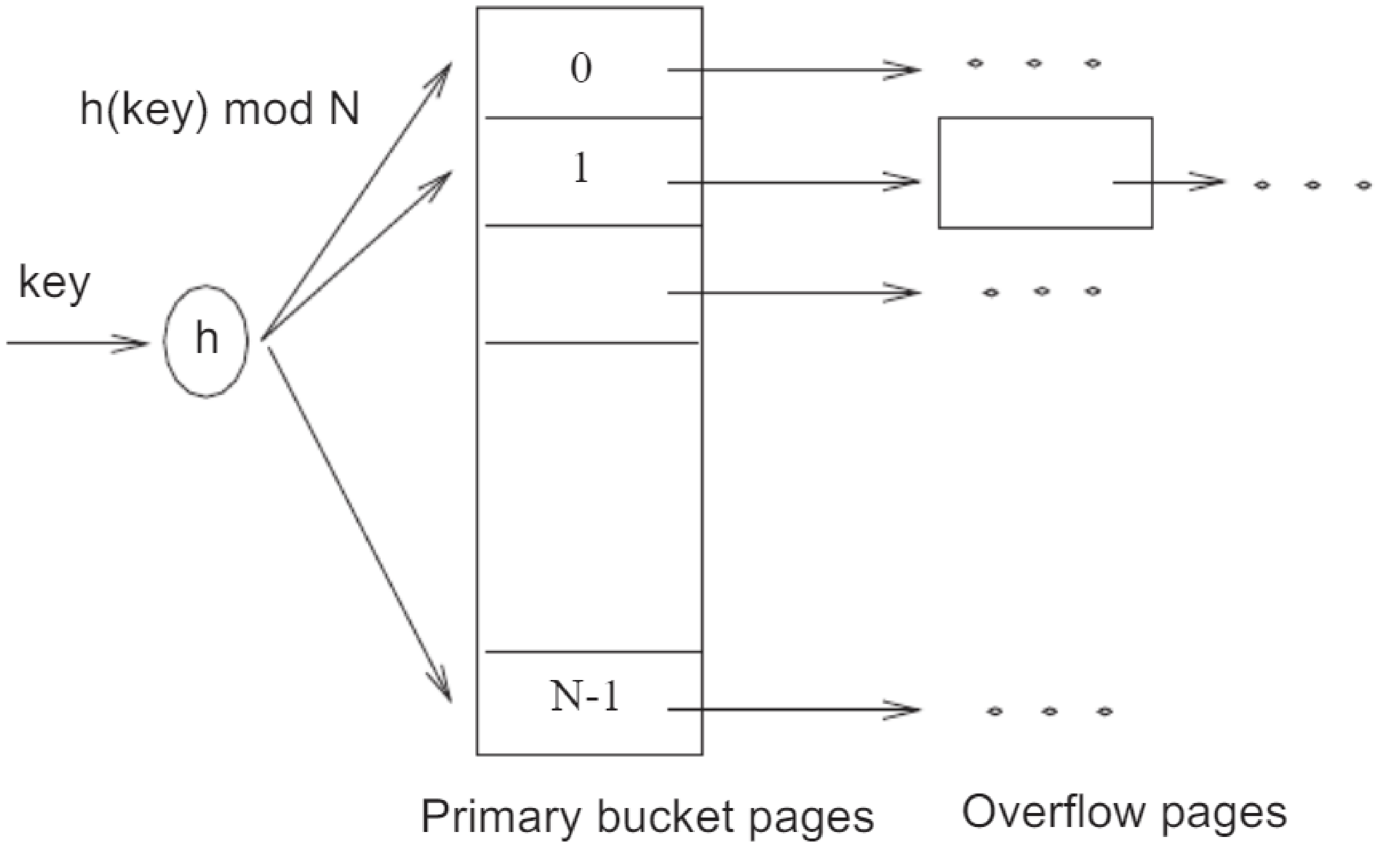
Hashing Schemes

1. **Static Hashing**, e.g., ISAM:
   - A simple scheme that maps values to a fixed bucket
   - Suffers from the problem of long overflow chains (collision), which can affect performance
   - Solution: Dynamic hashing, e.g., Extendible hashing & Linear hashing
2. **Extendible Hashing:**
   - Scheme uses a directory to support inserts & deletes efficiently with no overflow pages
3. **Linear hashing:**
   - Scheme uses a smart policy for creating new buckets & supports inserts & deletes efficiently without the use of a directory

Static Hashing

- Hash indexing where the # of buckets is fixed when the index is created
- Formally, let ***K*** denote the set of all search-key values & let ***B*** denote the set of all bucket addresses.
- A hash function ***h*** is a function from ***K*** to ***B***.
- To insert a record with search key $\boldsymbol{K_i}$, we compute $\boldsymbol{h(K_i)}$, which gives the address of the bucket for that record.
- The pages containing the data can be viewed as a collection of buckets, with one primary page & possibly additional overflow pages per bucket
- A file consists of buckets *0* through *N–1*, with one primary page per bucket initially
- Buckets contain the data entries

Static Hashing



Static Hashing Operations

- Hash function is used to locate entries for **search, insertion & deletion**
- **Search**: When a record needs to be retrieved, apply the hash function ***h*** to identify the bucket to which it belongs & then search this bucket
- **Insertion**: To insert a record, use the hash function ***h*** to identify the correct bucket & then put the data entry there

Bucket address = h(K)

- If there is no space for this entry,
  - Allocate a new overflow page, put the data entry on this page & add the page to the overflow chain of the bucket
- **Delete**: a search followed by a deletion operation

Bucket Overflow (Collision)

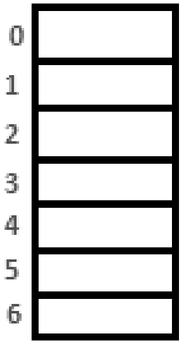
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.
- **Overflow Buckets:** Reorganize the files by doubling the # of buckets & redistributing the entries across the new set of buckets
- Suffers from one major defect—the entire file must be read & twice as many pages must be written, to achieve the reorganization

Handling of Bucket Overflows

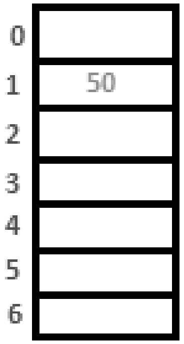
- **Closed addressing or Closed hashing or Overflow chaining:**
  - Occurs where records are stored in different buckets
  - Overflow buckets are chained together in a linked list
  - Compute the hash function & search the corresponding bucket to find a record
- **Open addressing:**
  - Occurs where all records are stored in **one** bucket
  - Use the available space in some other buckets
  - Uses linear probing to determine the next slot
  - Not used much in database applications

Open Addressing

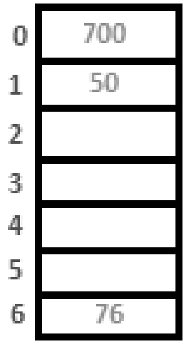
- Consider a simple hash function as “*key mod 7*”
- Sequence of keys as 50, 700, 76, 85, 92, 73, 101



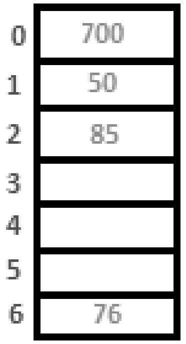
Initial Empty Table



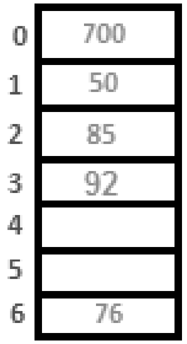
Insert 50



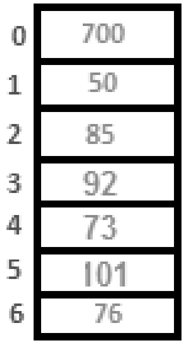
Insert 700 and 76



Insert 85: Collision Occurs, insert 85 at next free slot.

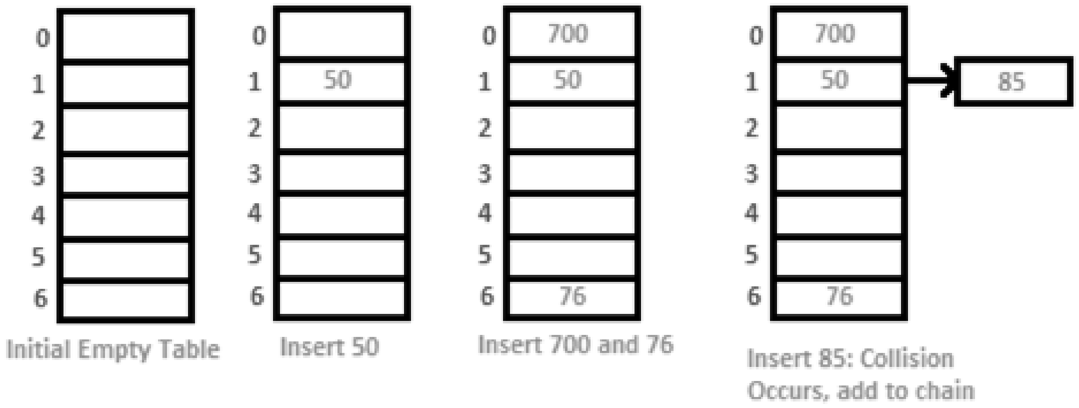


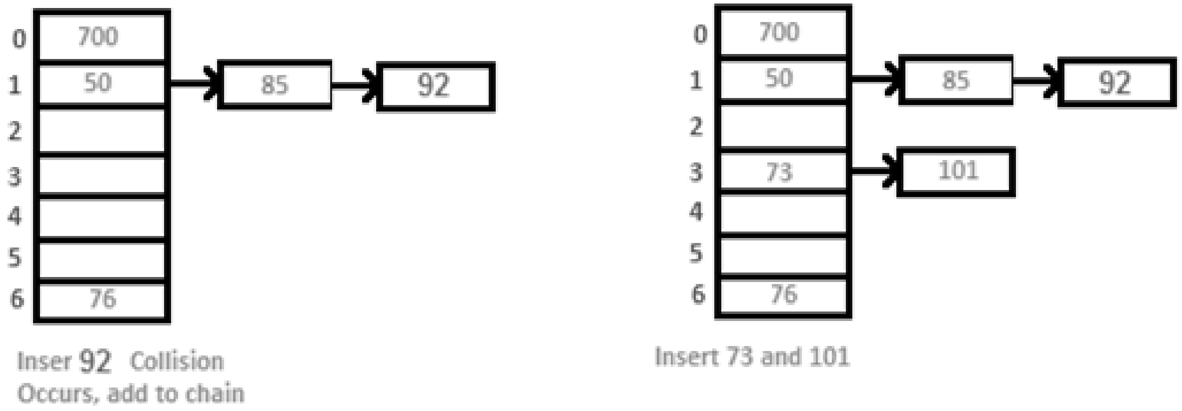
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot



Insert 73 and 101

Closed Addressing





Drawback of Static Scheme

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses
- # of bucket is fixed but databases may grow with time
  - If # is too small, we get too many ‘collisions’ resulting in records of many search key values being in the same bucket, degrading performance
  - If # is too large, wastes space
- **A Solution:**
  - Periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- **Better solution:**
  - Allow the # of buckets to be modified dynamically

Dynamic Hashing

- Schemes proposed that allow the # of buckets to be increased in a more incremental fashion
- **Periodic Rehashing**
  - E.g., If number of entries in a hash table becomes, e.g., 1.5 times size of hash table, create new hash table of size 2 times the size of previous hash table. Rehash all entries to new table.
- **Linear Hashing:**
  - Rehashing in an incremental manner
- **Extendible Hashing**
  - A mechanism in which data buckets are added & removed dynamically & on-demand

Extendible Hashing

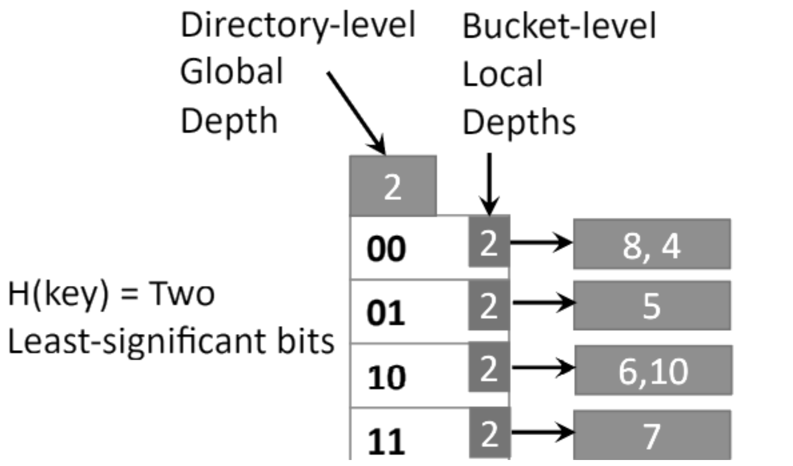
- A dynamically updateable disk-based index structure, which implements a hashing scheme utilizing a directory
- Designed to minimize the cost of rehashing
- Use least significant bits (LSB) of a key to hash the key into a bucket
- **Directories:** Containers that store pointers to buckets
- **Buckets:** Store the hashed keys
- **Local Depth:** # assigned per bucket. (# of bits used to hash data into bucket)
- **Global Depth:** Maximum # of bits used to hash data to any of the buckets
- Initially, Local Depth = Global Depth for all bucket

Extendible Hashing Steps

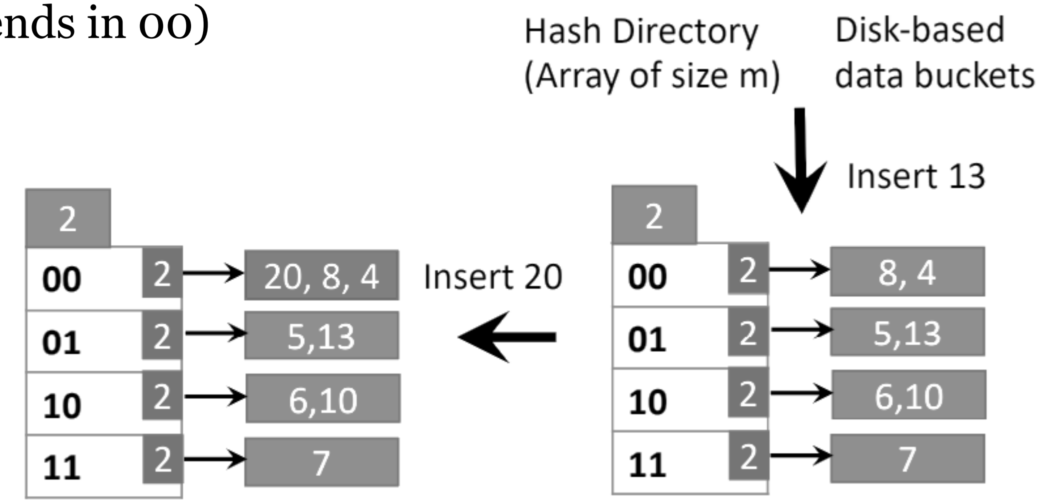
1. Analyze data elements, e.g., int, string, etc.
2. Convert to binary format, e.g., 49 is 110001. For strings use ASCII equivalent
3. Check Global depth of the directory, e.g., suppose global depth is 3
4. Identify the directory, e.g., for 110**001**, the LSB is 001 for a global depth of 3
5. Navigate to the bucket pointed by the directory with id-001
6. Insertion & Overflow check, else end if no overflow
7. If an overflow occurs,
    1. Case 1: if global depth = local depth, then expand directory, split bucket & increment global depth by 1
    2. Case 2: if local depth < global depth, split bucket & increment local depth value by 1
8. Rehash of split bucket elements

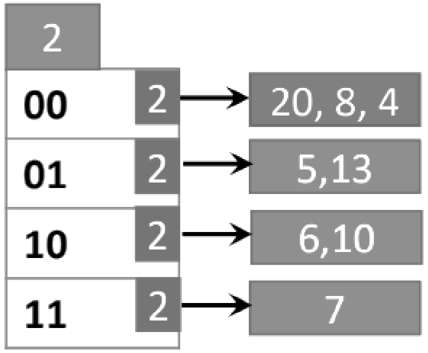
Example: Insertion

- Assuming the bucket capacity for each bucket page is 2
- **Insert 13** (Binary: 11**01** – ends in 01)
- Look to only 2 bits because Global Depth = 2
- **Insert 20** (Binary: 101**00** - ends in 00)
- Result in bucket overflow:
  1. Double the directory
  2. Increment Global Depth by 1
  3. Rehash only the overflowed bucket
  4. Increase this bucket's local depth



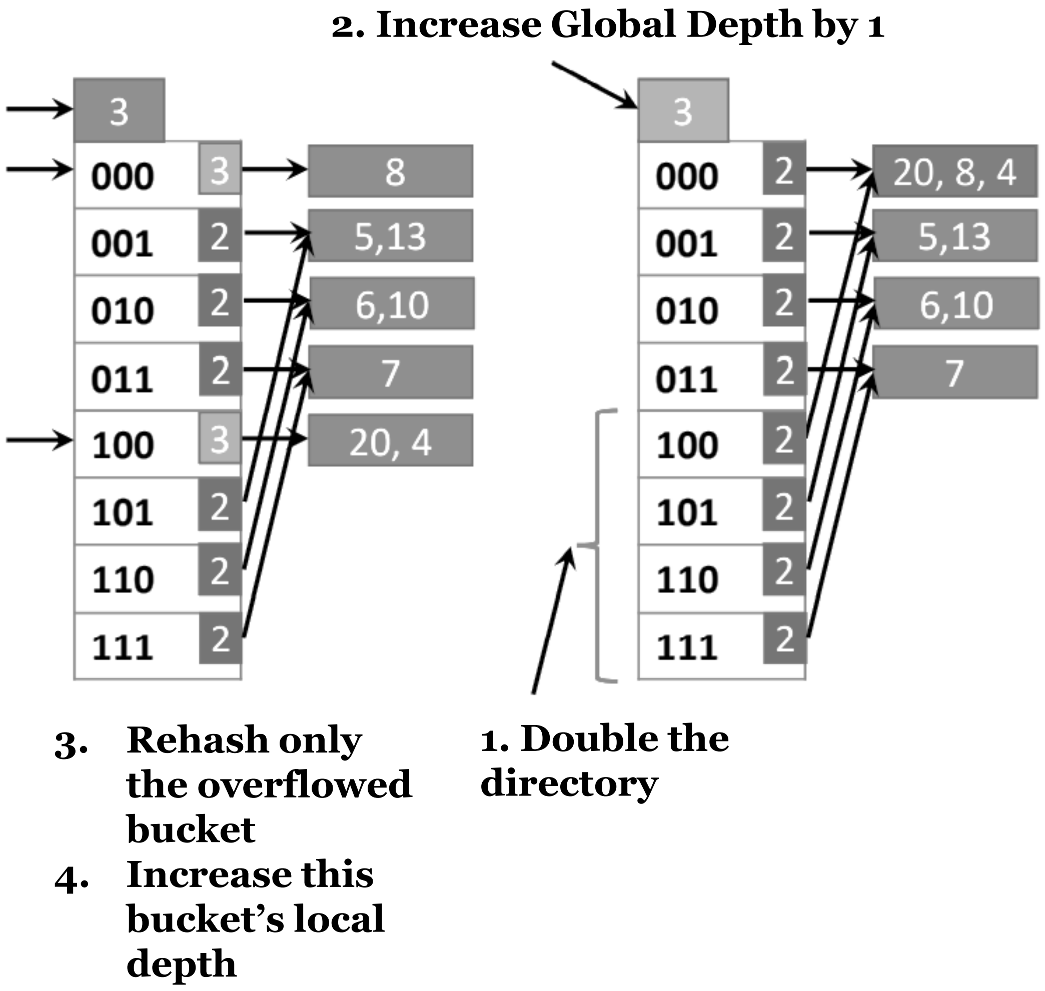
Directory-level Global Depth
Bucket-level Local Depths
H(key) = Two Least-significant bits
2
00
01
10
11
2
2
2
2
8, 4
5
6,10
7





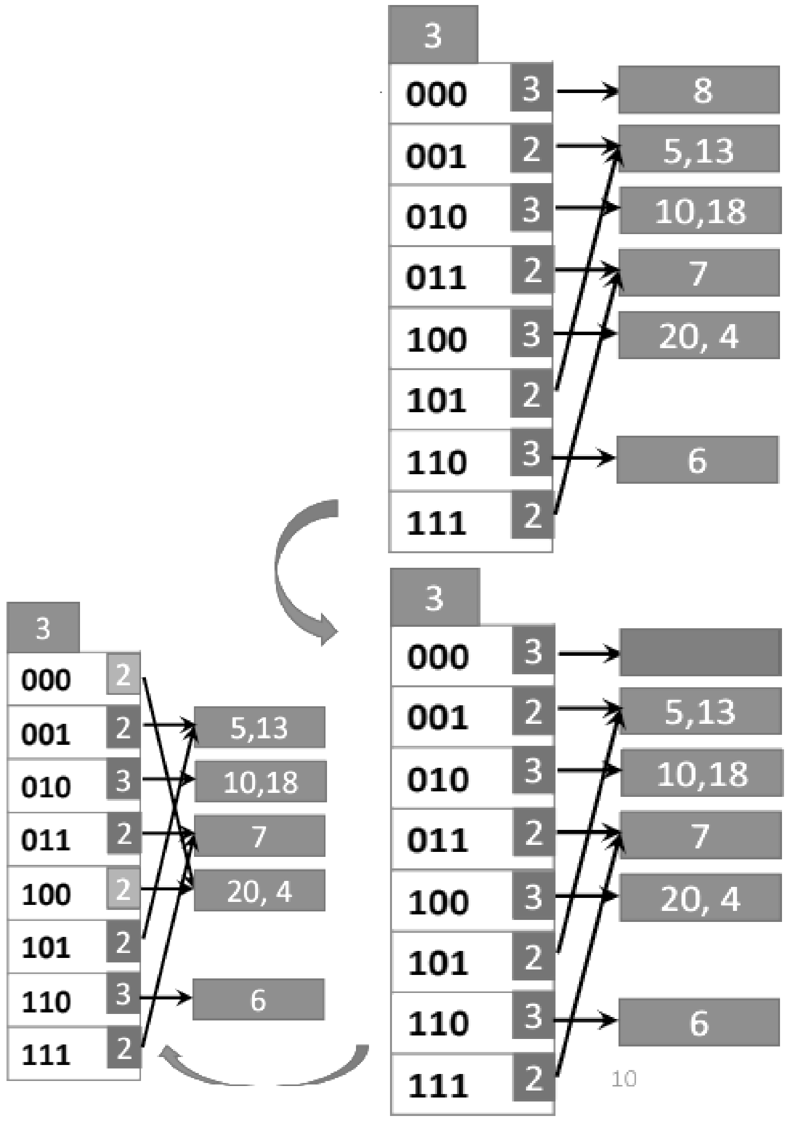
- Insert 20 (Binary: 10100 - ends in 00)
- Results in bucket overflow:

1. Double the directory
2. Increase Global Depth
3. Rehash only the overflowed bucket
4. Increase this bucket's local depth



Example: Deletion

- Delete 8 (1000 Ends in 000)
  - Results in an empty bucket
  - Delete this bucket
  - Merge Bucket 000 with Bucket 100 into 1
  - Make them point to the same bucket
  - Decrease both their Local Depths by 1



Multiple Key Access

Multiple Key Access

- Use multiple indices for certain types of queries.
- Example:

  SELECT *ID*

  FROM *instructor*

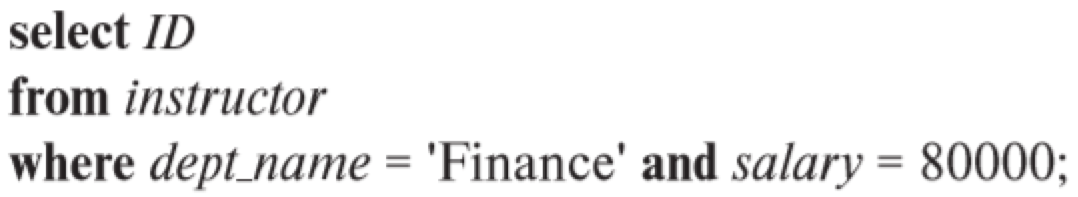
  WHERE *dept_name* = "Finance"

  AND *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  1. **Use index on *dept_name*** to find instructors with department name Finance; test *salary = 80000*
  2. **Use index on *salary*** to find instructors with a salary of $80000; test *dept_name* = "Finance".
  3. **Use *dept_name* index** to find pointers to all records pertaining to the "Finance" department.
  4. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Index on Multiple Keys

- An alternative strategy for this case is to create & use an index on a composite search key *(dept name, salary)*
- **Composite search keys** are search keys containing more than one attribute, E.g., (*dept_name, salary*)
- Ordered (B+ tree) index can be used on the preceding composite search key to answer efficiently queries of the form:



```
select ID
from instructor
where dept_name = 'Finance' and salary = 80000;
```

- Queries which specifies an **equality condition** on the first attribute & range on second attribute can be handled

Index on Multiple Keys

- Suppose we have an index on combined search-key (*dept_name, salary*) with the **where** clause

  where *dept_name* = "Finance" **and** *salary* = 80000

- The index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
- Can also efficiently handle with range on second attribute

  where *dept_name* = "Finance" **and** *salary* < 80000

- But cannot efficiently handle

  where *dept_name* < "Finance" **and** *balance* = 80000

  - May fetch many records that satisfy the first but not the second condition

Bitmap Index

- A bitmap index is a special type of index designed for efficient query processing on multiple keys
- It is a **binary valued 2D array** created with an indexed column for every record in the table
- Records in a relation are assumed to be numbered sequentially
- Applicable on attributes that take on a relatively small # of distinct values.
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small # of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- Bitmap indexes use bit arrays (commonly called **bitmaps**) & answer queries by performing bitwise logical operations on these bitmaps

Bitmap Index

- Simply, bitmap index on an attribute has a **bitmap for each value of the attribute**.
  - The bit in a row of bitmap is “**1**” if the record has the value v for the indexed attribute, or “**0**” otherwise
- <u>Example:</u>

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Index

- Bitmap indexes are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size & applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000

Bitmap Index

- **Advantages:**
  - Reduced storage requirements compared to other indexing techniques
  - Bitmap indices are generally very small compared with relation size
  - Reduced response time for large classes of ad-hoc queries
  - Multiple indices can be combined for executing a query

SQL Index Definition

- Syntax:

```
CREATE INDEX <index-name>
ON <relation-name>(<attribute-list>)
```

E.g.,: **CREATE INDEX** *b-index* **ON** *branch(branch_name)*

- Use **CREATE UNIQUE INDEX** to indirectly specify & enforce the condition that the search key is a candidate
- Not really required if SQL **UNIQUE** integrity constraint is supported
- To drop an index

```
DROP INDEX <index-name>
```

Creation of Indices

CREATE INDEX *takes_pk*
    ON *takes* (*ID,course_ID, year, semester, section*)

DROP INDEX *takes_pk*

- Most database systems allow **specification of type of index & clustering**
- **Indices on primary key** are created automatically by all databases
- Some database also create **indices on foreign key attributes**
- Indices can greatly speed up lookups, but impose cost on updates