Chapter 11: Indexing

Database System Concepts, 6th Ed.

Basic Concepts

- Indexing mechanisms used to speed up access to desired data
  - E.g., author catalog in library
- **Search Key** - attribute used to look up records in a file
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

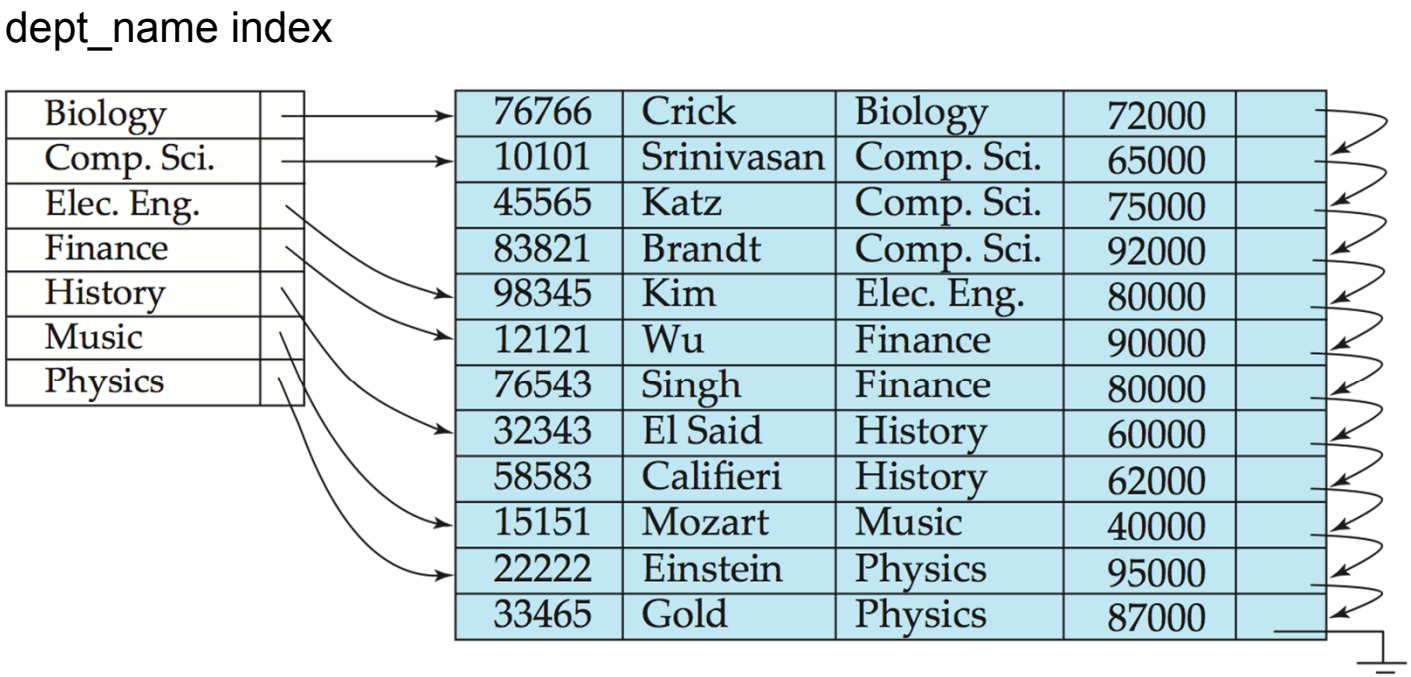
Index Evaluation Metrics

- The types of access that are supported efficiently
  - Finding records with a specified value in the attribute
  - Finding records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

Ordered Indices

- **Ordered index:** entries in the index file are stored, sorted on the search-key value. The search key value need not be the primary key of the file
- Example. Search-key is *dept_name*; Primary key is *ID*



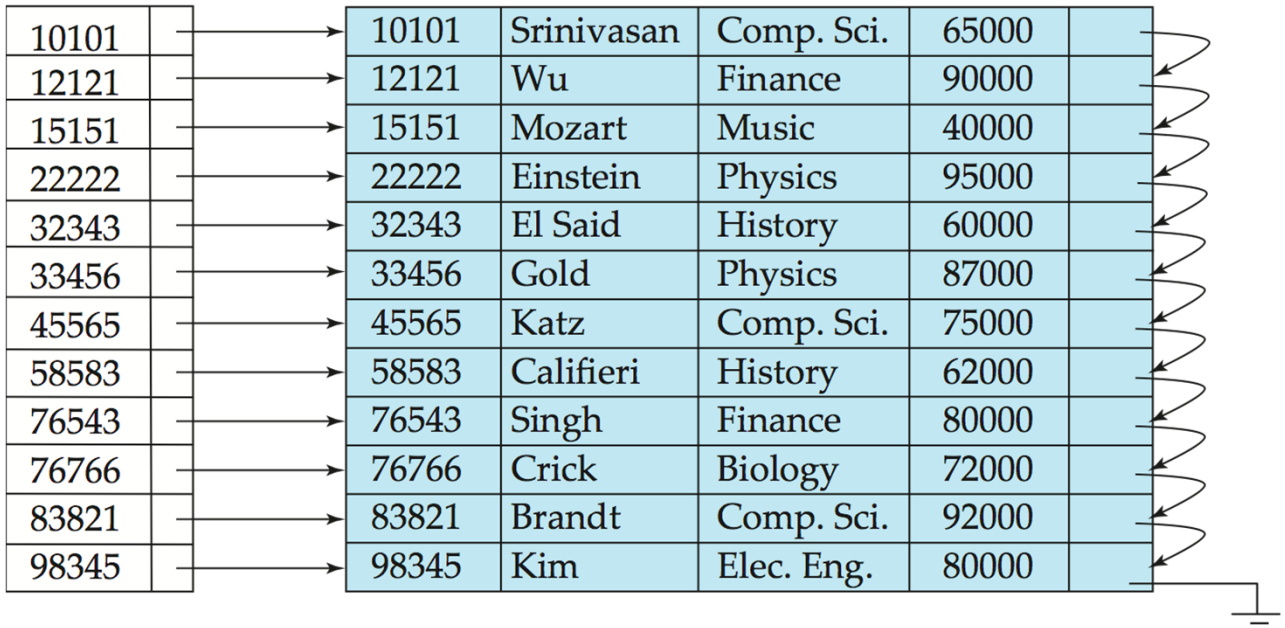
Ordered Indices (Cont.)

- **Clustering index:** an ordered index whose search key also defines the sequential order of the file. Also called **primary index**
  - The search key of a clustered index is **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**
  - Example:
    - File sequentially ordered on ID
    - Secondary index ordered on salary

Index-sequential File

- An ordered sequential file with a clustering index is called an **index-sequential file.**
- Example:
  - File is sequentially ordered on ID
  - Clustering index on ID

sequence of the index is the same as sequence of the file



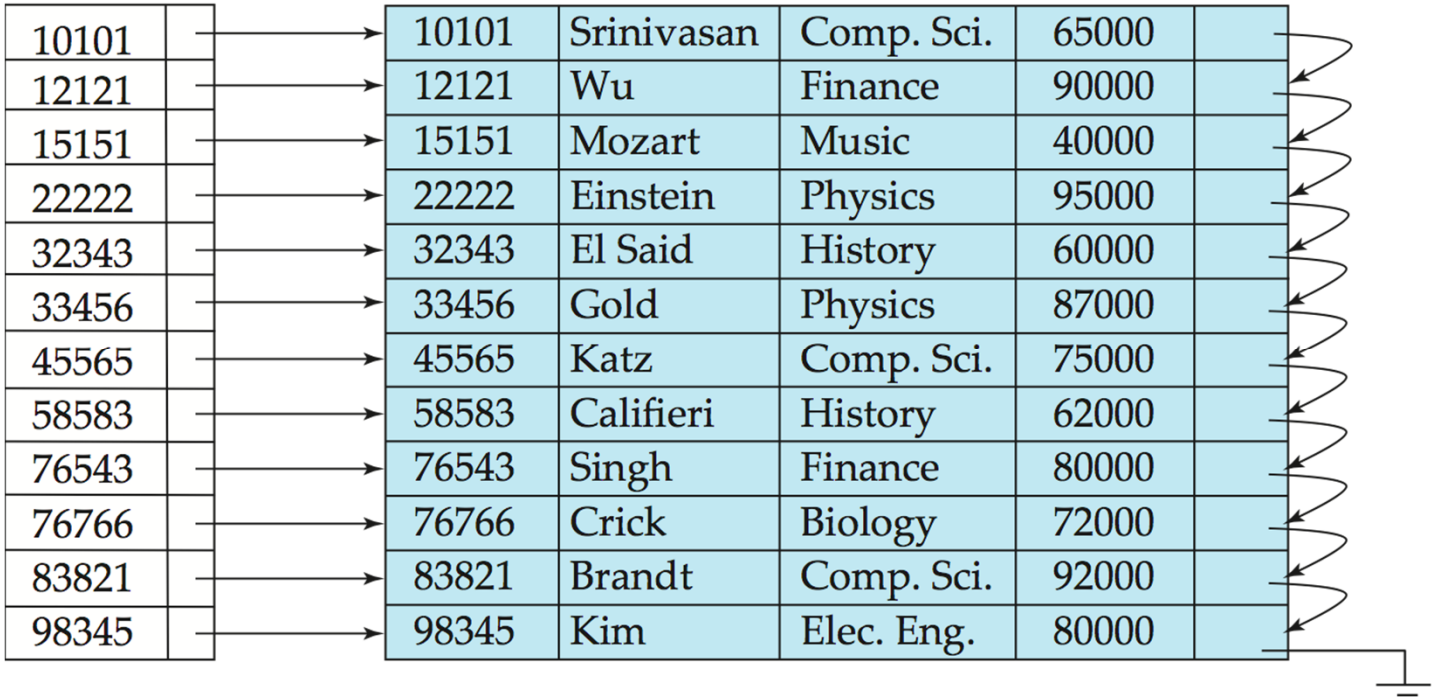
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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

Dense and Sparse indices

- There are two types of ordered indices
  - **Dense index,** Index record appears for every search-key value in the file
  - **Sparse index:** Index record appears for only some of search-key values in the file

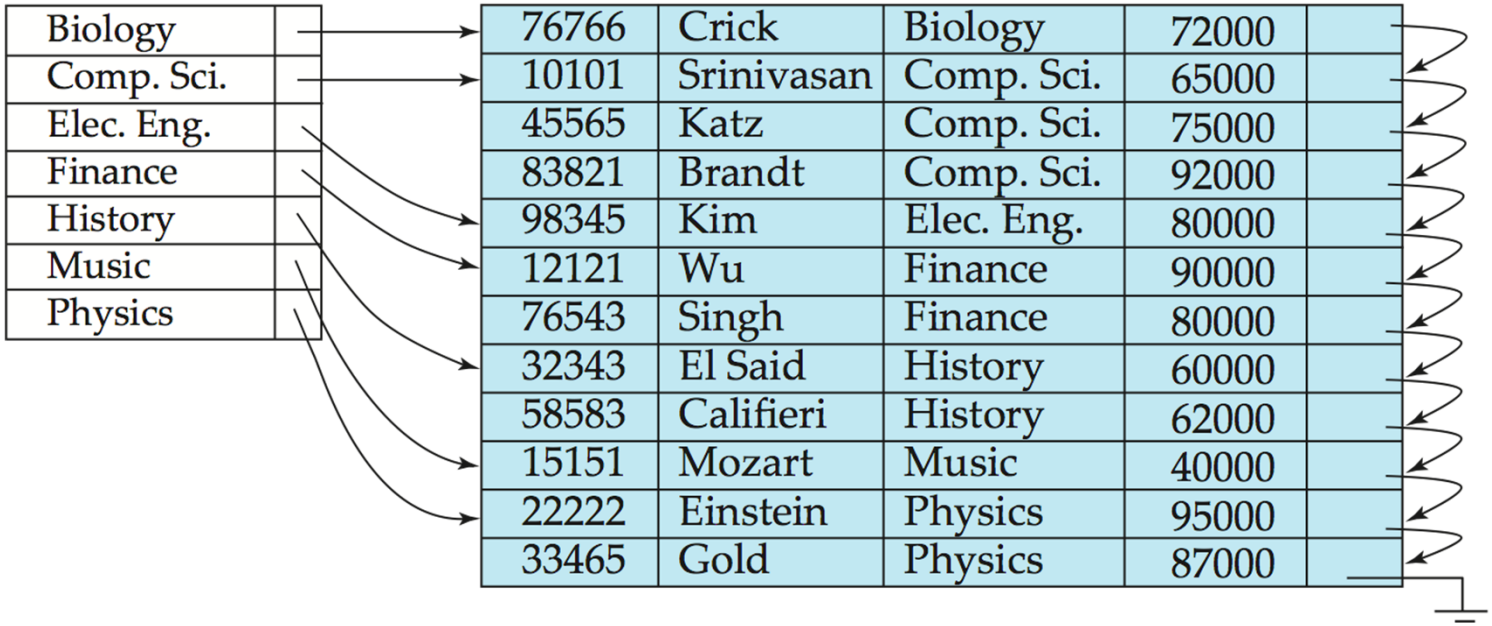
Dense Index Files

- Dense index on ID, with instructor file sorted on ID



Dense Index Files (Cont.)

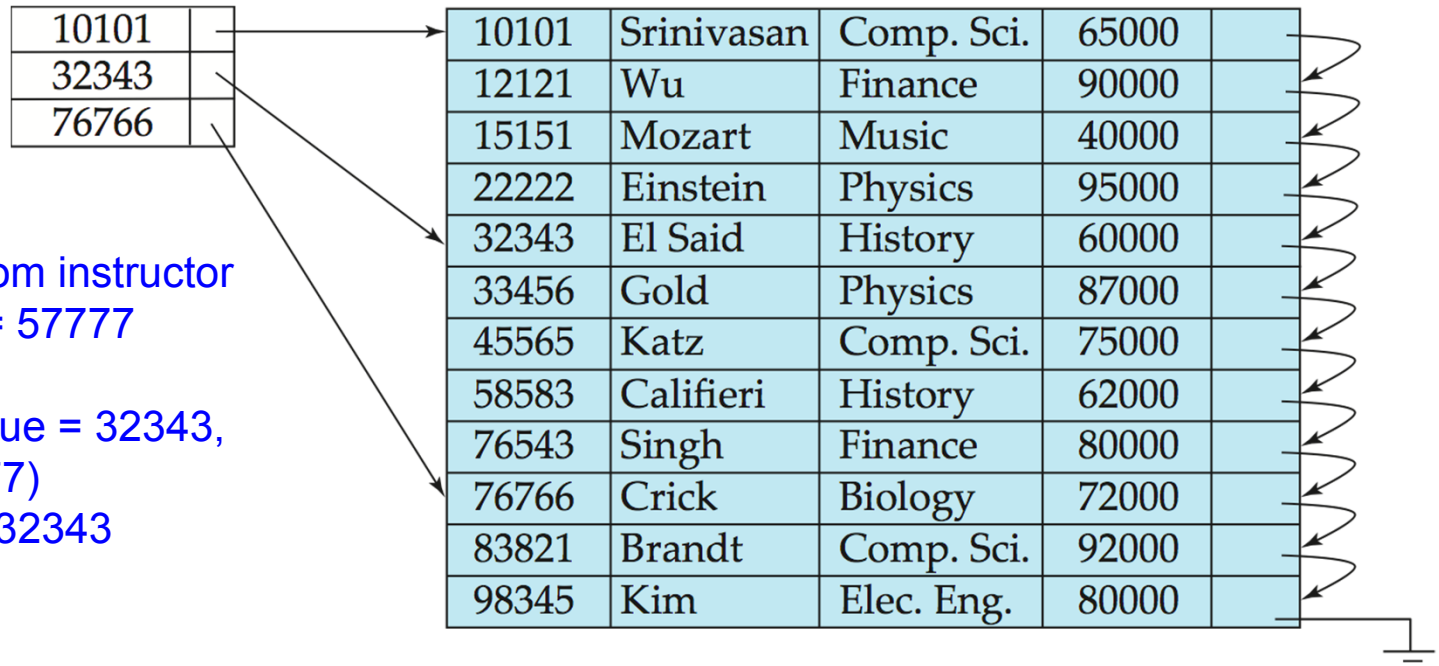
- Dense index on dept_name, with instructor file sorted on dept_name



even though dept_name is not primary key, still dense index all dept_name is in the index table

Sparse Index Files

- Applicable only if the relation is stored in sorted order of the search key; that is, if the index is a clustering index.
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

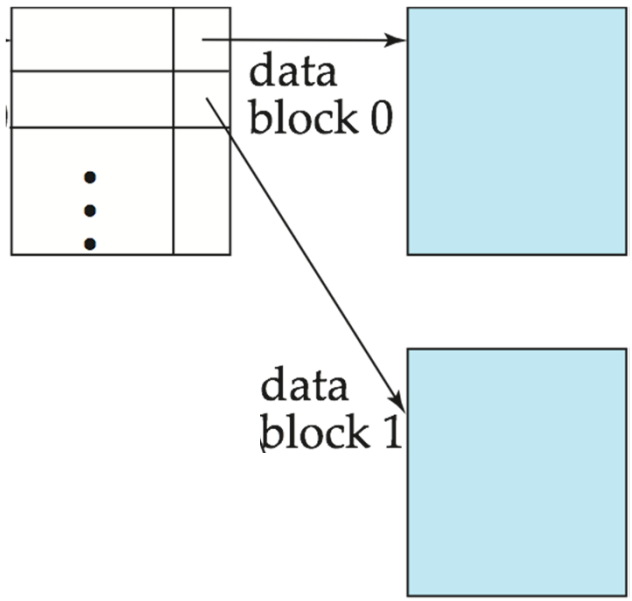


ex) select * from instructor
where ID = 57777

search key value = 32343,
value < K (5777)
point starts at 32343

Sparse Index Files (Cont.)

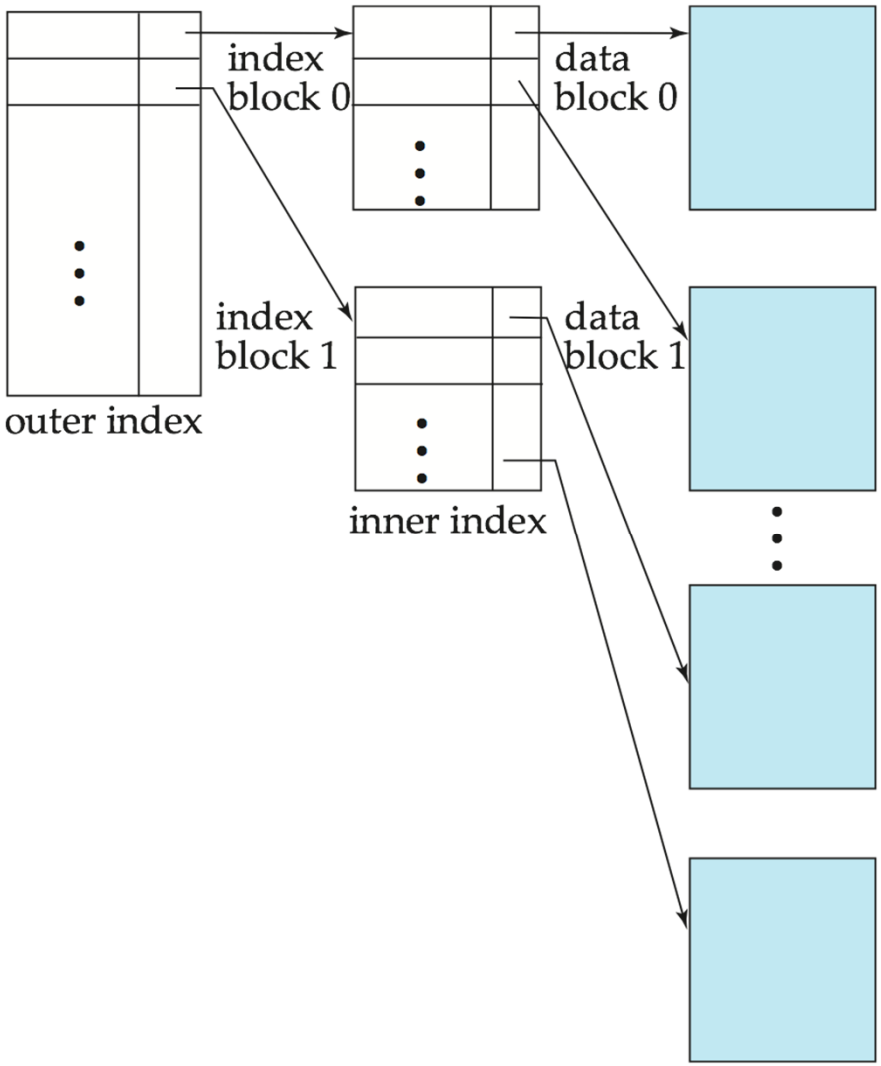
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Multilevel Index

- If the primary index does not fit in memory, access becomes expensive
- Solution: Keep the primary index on disk and treat it as a sequential file and construct a sparse index on it. The sparse index is kept in memory:
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index Example



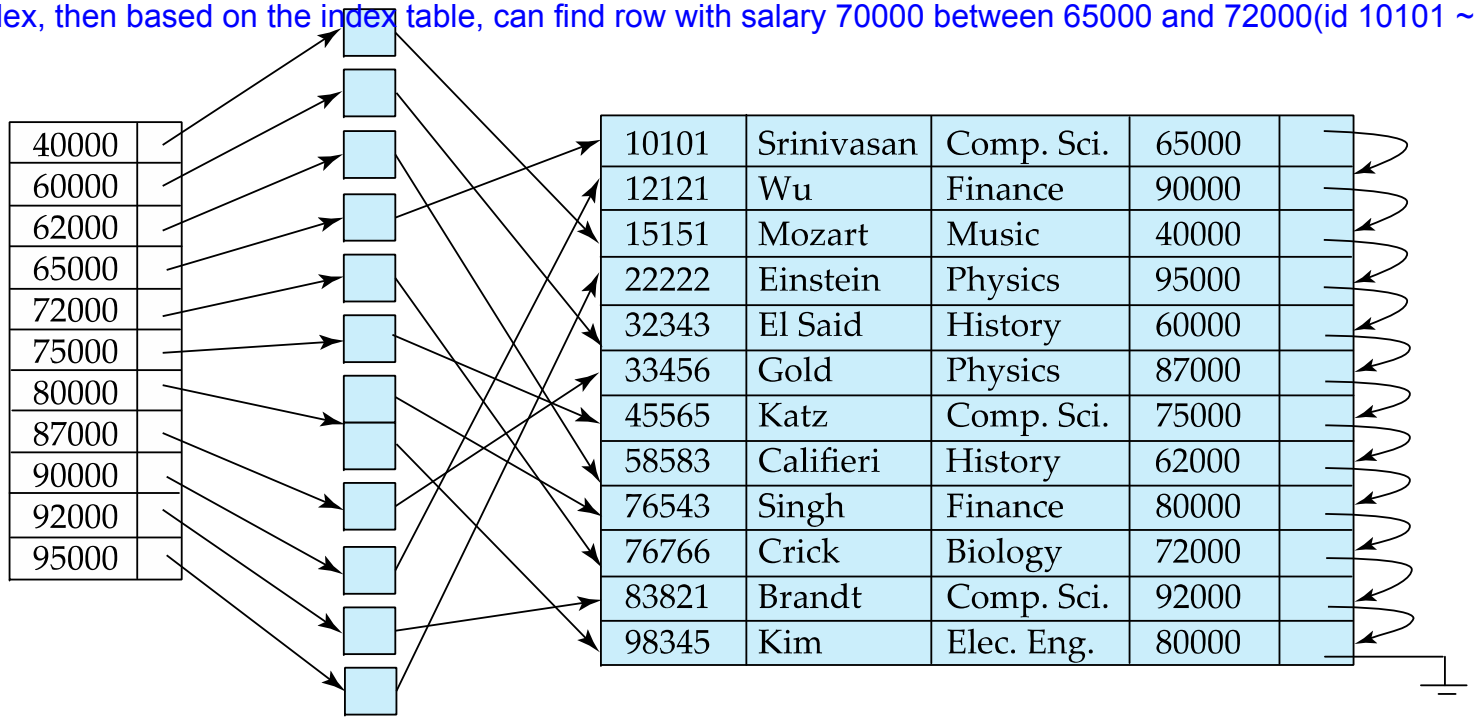
Secondary (nonclustering) Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition
  - Example 1: In the instructor relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Secondary Indices Example

- 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
- Example of secondary index on salary field of instructor

ex) get salary 70000!
if no secondary index, then look through all table rows as the main index is id
But with secondary index, then based on the index table, can find row with salary 70000 between 65000 and 72000(id 10101 ~ 76766)



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records
- **BUT**: Updating indices imposes overhead on database modification - when a file is modified, every index on the file must be updated
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

records with primary key stored together, efficient
but records with secondary index stored randomly, expensive

Disadvantage of Using index-sequential Files

- Performance degrades as file grows, both for index lookups and for sequential scans through the data.
- Since many overflow blocks get created over time, periodic reorganization of entire file is required.
- Solutions:
  - B^+-tree index files
  - Hashing

B^+-Tree Index Files

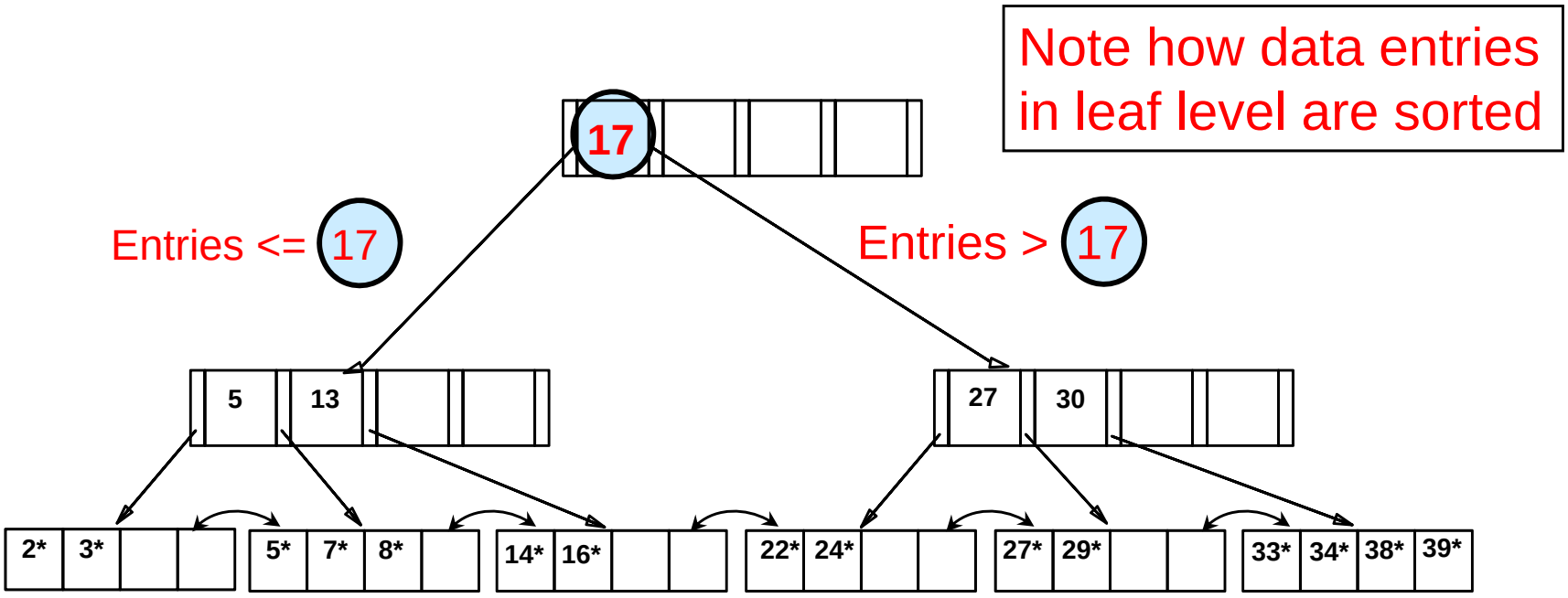
B^+-Tree Index Files

- The B^+-tree structure index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data
- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - Extra insertion and deletion overhead, space overhead
- History. What is the B standing for?
  - Bayer (who co-invented)
  - Boeing (the company that Bayer worked for)
  - Balanced

Tree Index

- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have index entries; only used to direct searches
- Good for range and equality selection (>,>=, =,<,<=,<>)

Tree Indexes



Hashing

Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block)
- In a **hash file organization** we obtain the bucket of a record directly 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 records for access, insertion as well as deletion
- Records with different search-key values may be mapped to the same bucket; thus, entire bucket has to be searched sequentially to locate a record

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key (See figure in the next slide)

- There are 8 buckets,
- Assume that the i^{th} letter in the alphabet is represented by the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 8
  - E.g. h(Music) = 1

    h(History) = 2

    h(Physics) = 3

    h(Elec. Eng.) = 3

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

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

Hash Functions

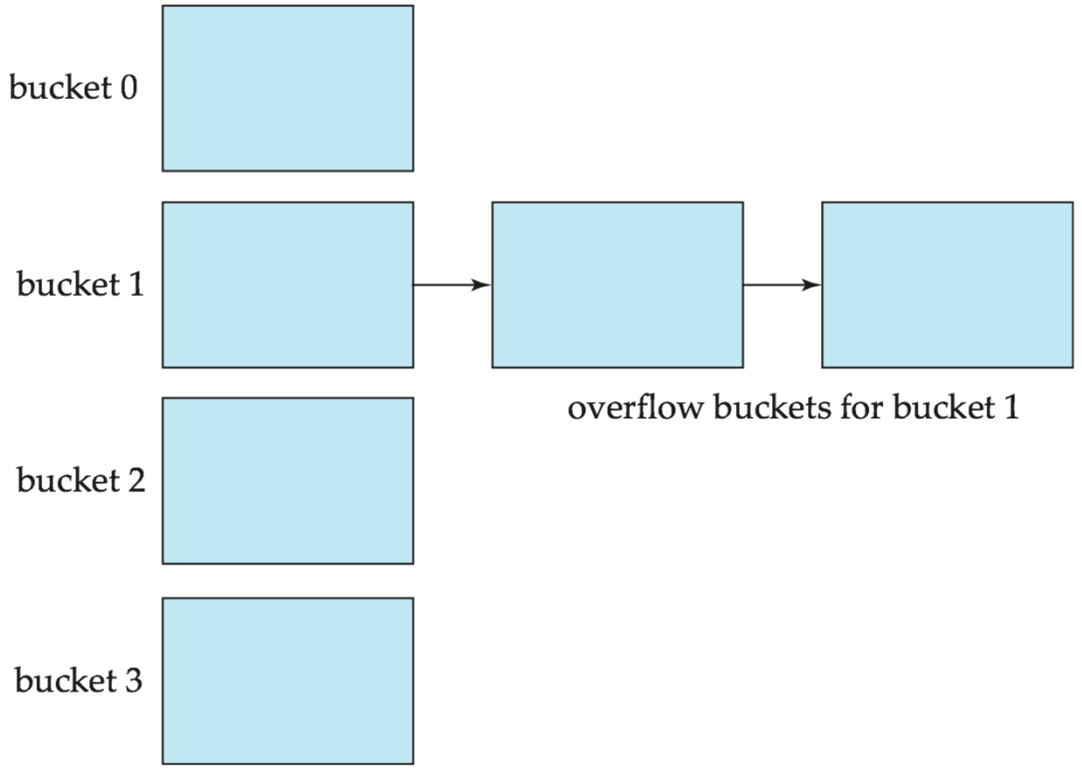
- Hash functions require careful design
  - A bad hash function may result in look up taking time proportional to the number of search keys in the file, e.g., most records with the same hash value
  - A good hash function "uniformly" distributes the data across the entire set of possible hash values. The hash function generates very different hash values for similar strings (random, no correlation between search keys and hash values)

Bucket Overflows

- When we need to store a search-key value in a bucket and that bucket is full, we must store the data in an ***overflow bucket***
- Bucket overflow can occur because of
  - Insufficient number of 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

Handling of Bucket Overflows (Cont.)

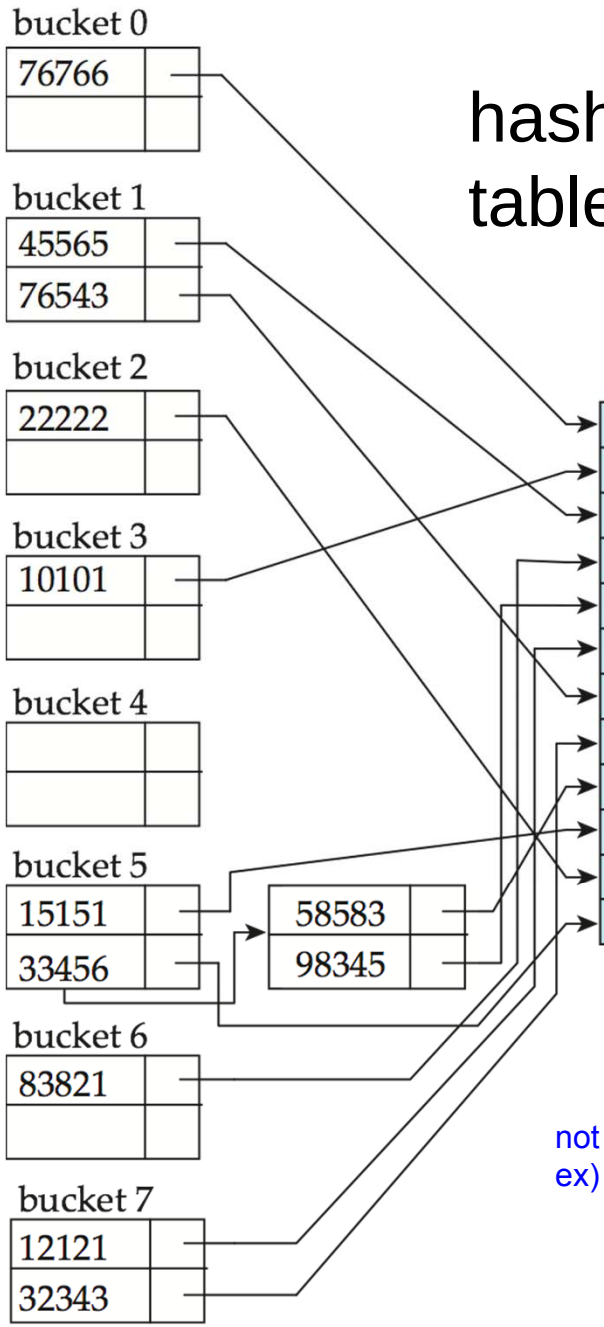
- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list. This scheme is called **closed hashing**



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure
- Strictly speaking, a hash index is always a secondary index
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
  - However, we use the term hash index to refer to both secondary index structures and hash organized files

Example of Hash Index



hash index on attribute *ID* of the i*nstructor* table, which is ordered by *dept_name*

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

not good for range check
ex) 12121 is in bucket 7 but the next index which is 5151 in bucket 5

Comparison of Ordered Indexing and Hashing

- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - MySQL uses both B-tree and hash indexes
  - Oracle supports static hash organization, but not hash indices
  - SQL Server supports only B-trees
- Most DBMSs automatically indexed PK

Creating Indexes with SQL Server Management Studio

- https://www.mssqltips.com/sqlservertip/5573/creating-indexes-with-sql-server-management-studio/