Indexing 1

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

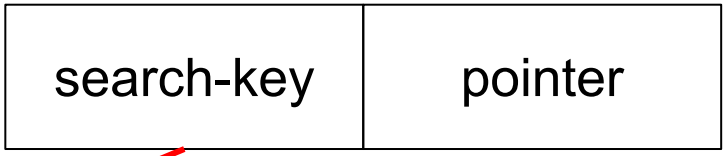
Department of Computer Science

Overview

- Basic Concepts
- Ordered Indices
- Assignments

Basic Concepts

- Indexing mechanisms are used to speed up access to desired data (e.g. author catalog in library)
- An **index file** consists of records (called **index entries**) of the form



- **Search Key** is an attribute or set of attributes used to look up records in a file
  - Note that it is different from primary key, candidate key and superkey
- Index files are typically much smaller than the original file
- An index file may store several indices on several search keys

Basic Concepts

- 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”
- There are several techniques for ordered indexing
  - No one technique for indexing is the best.
  - Each technique must be evaluated on the basis of the following factors:
    - Access types supported efficiently. E.g.,
      - Records with a specified value in the attribute
      - Records with an attribute value falling in a specified range of values
    - Access time
    - Insertion time
    - Deletion time
    - Space overhead

Overview

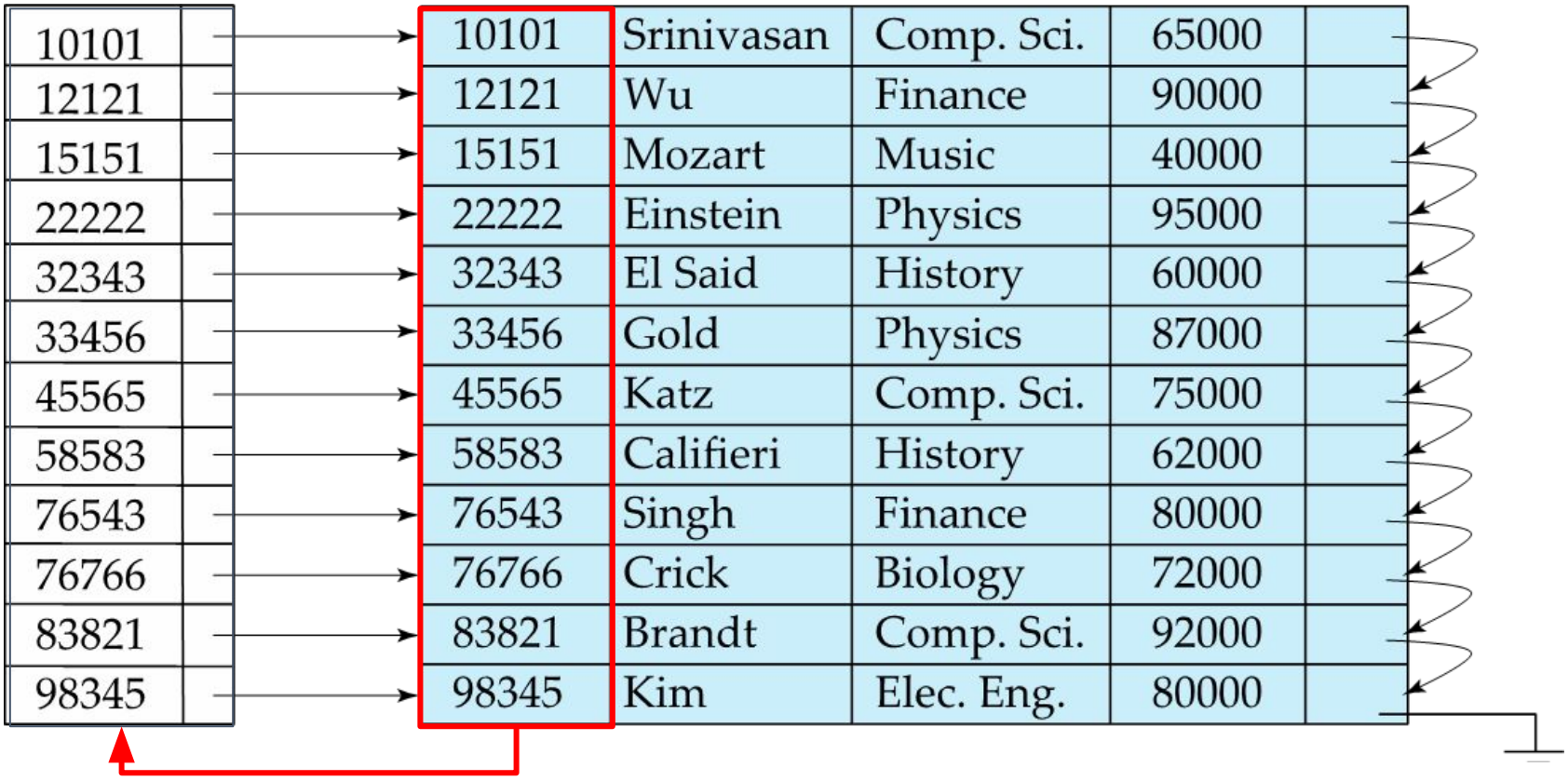
- Basic Concepts
- Ordered Indices
- Assignments

Ordered Indices

- In an **ordered index,** index entries (also called index records) are stored sorted on the search key value
  - e.g. the index of a book or a library catalog
- **Clustering index:** in a sequentially ordered file, the 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
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file
  - Also called **nonclustering index**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key

Dense Index Files

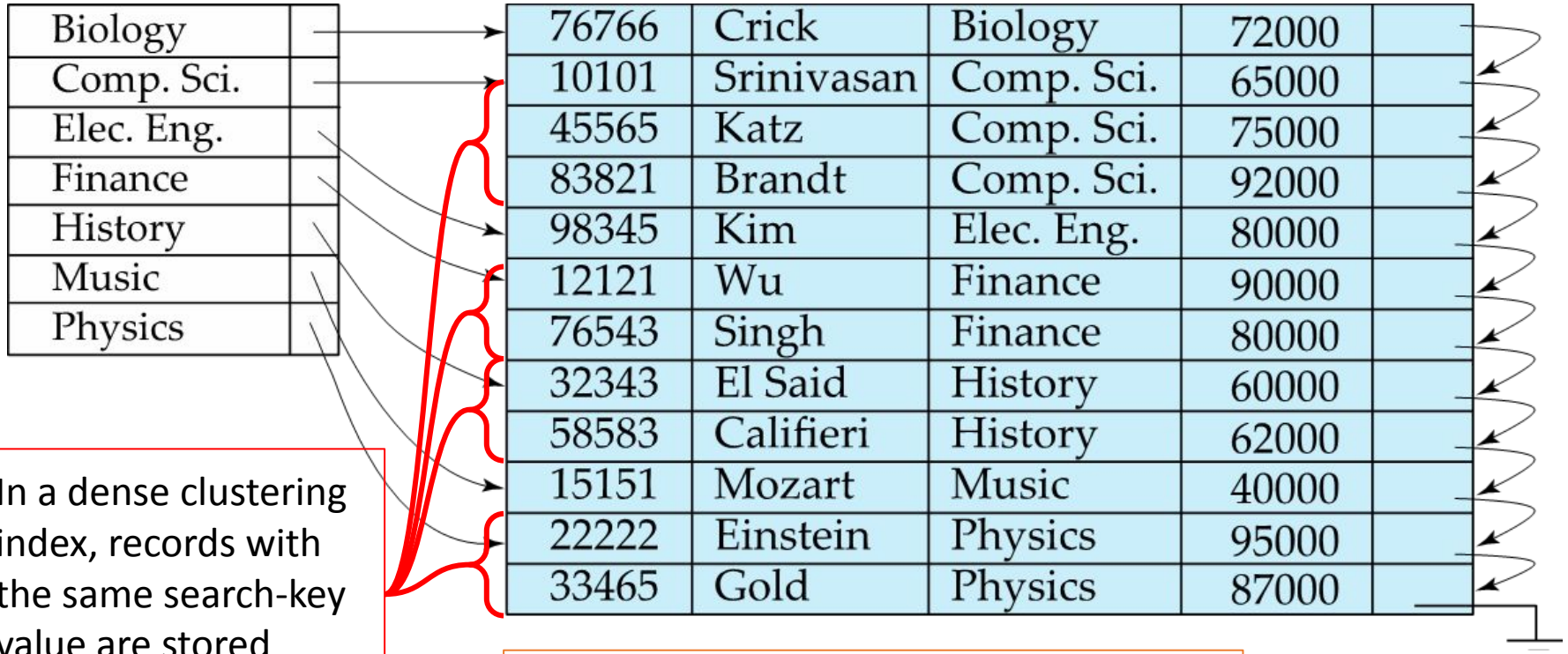
- **Dense index** - the index entry appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



For every search-key value, there is the corresponding index entry in the index file

Dense Index Files (Cont.)

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

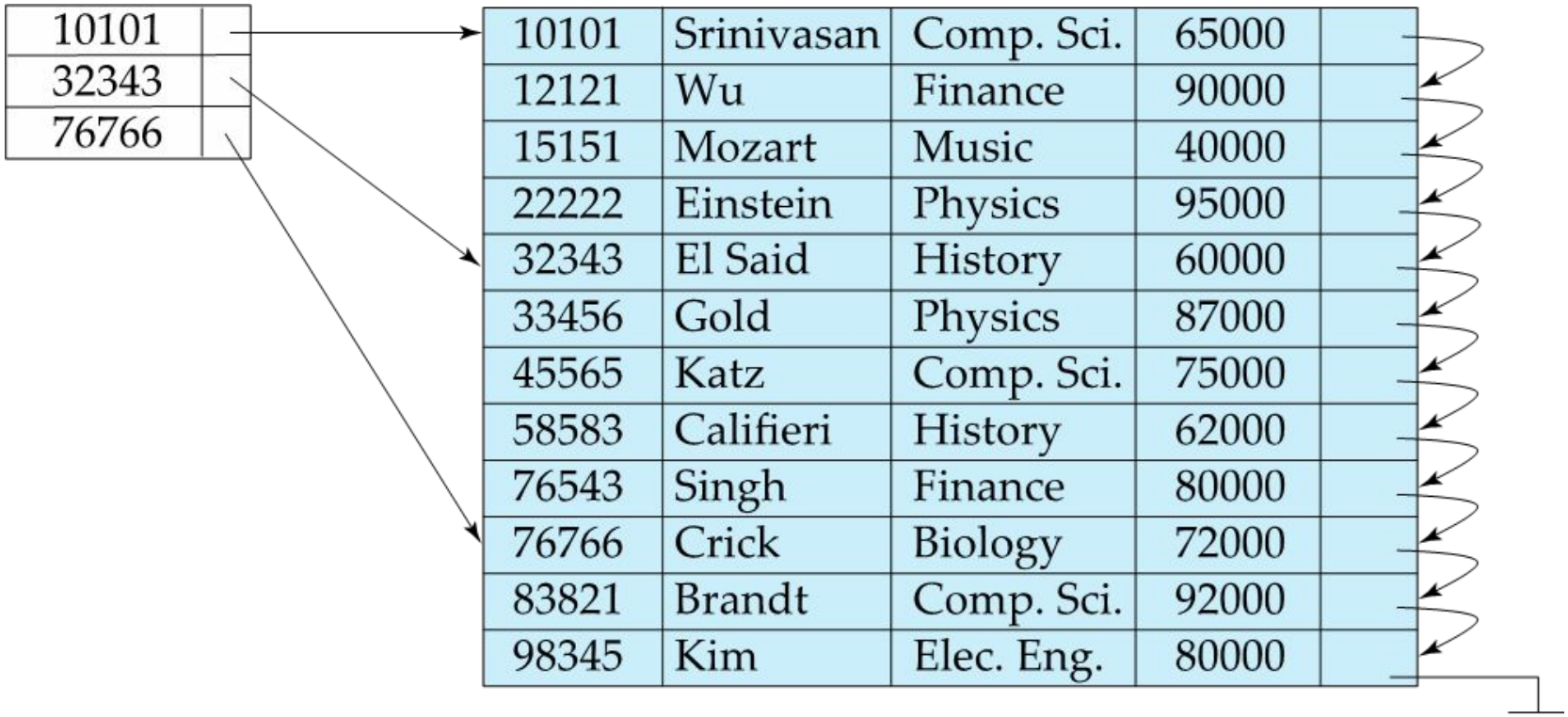


In a dense clustering index, records with the same search-key value are stored sequentially

In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

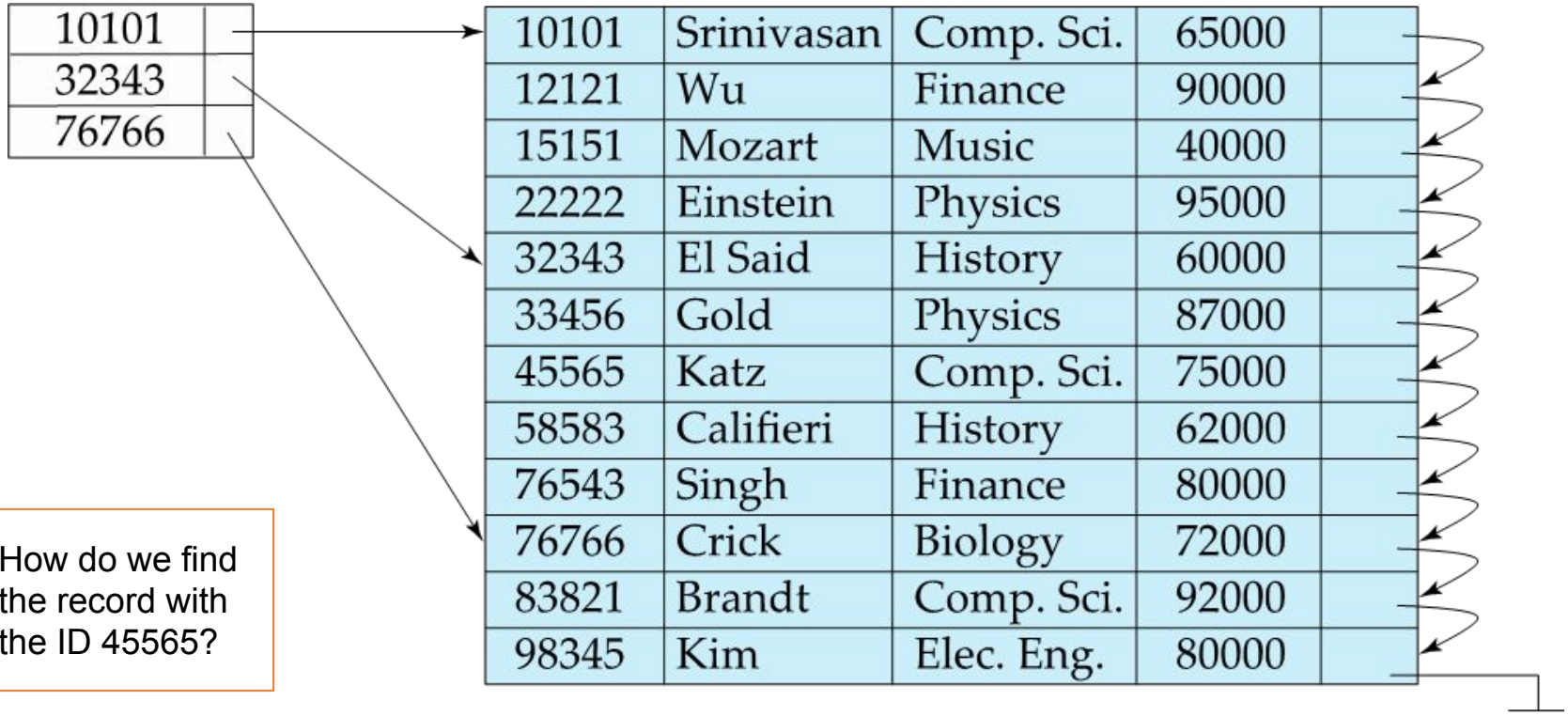
Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when relation is stored in sorted order of the search-key
  - That is, sparse index can be used if the index is a **clustering index**



Sparse Index Files

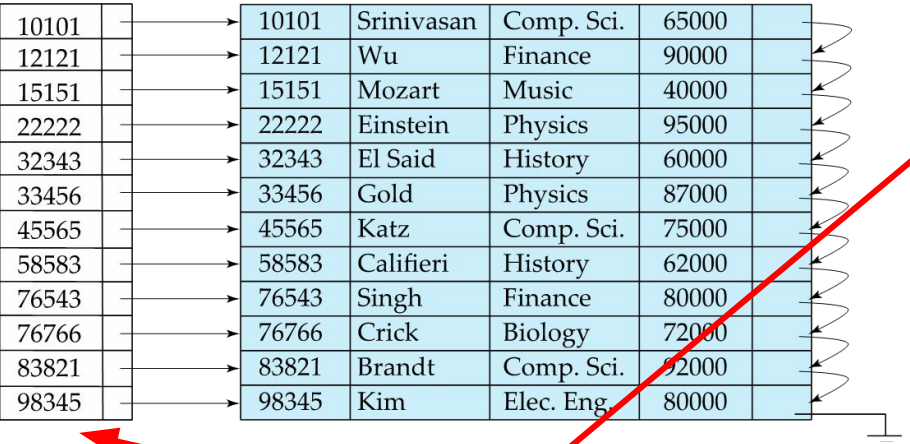
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

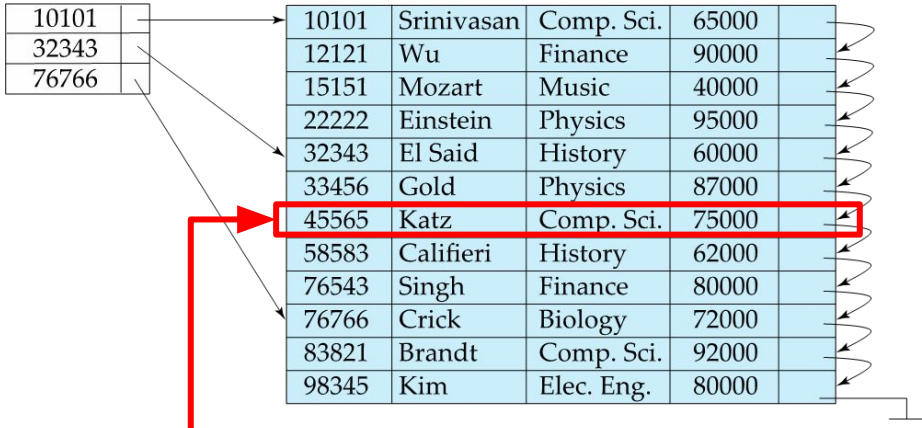


How do we find the record with the ID 45565?

Sparse Index Files (Cont.)

- Compared to dense indices:





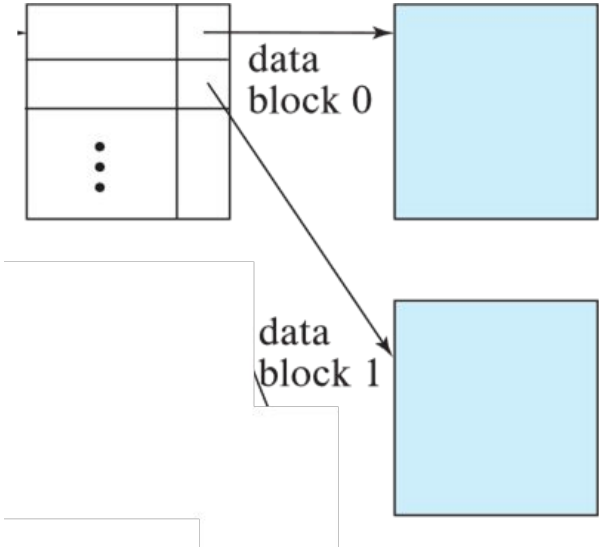
Less space and less maintenance overhead for insertions and deletions.

Generally slower than dense index for locating records.

There is a trade-off between access time and space overhead

Sparse Index Files (Cont.)

- **Good tradeoff**:
  - sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



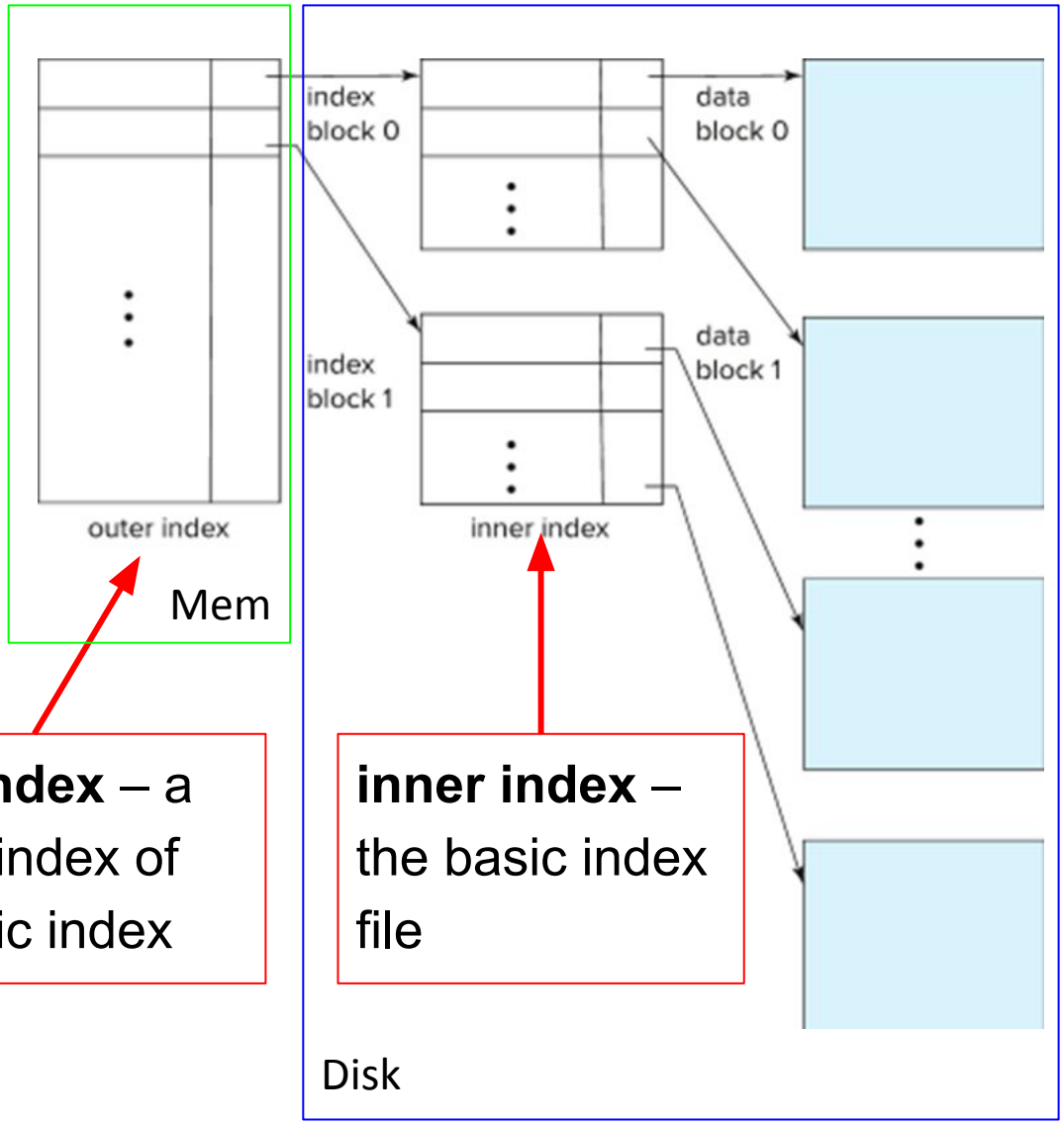
Significantly reduce disk access time while maintaining less space overhead

Multilevel Index

- If index does not fit in memory, access becomes expensive.
  - Suppose we are building an dense index for a relation that has 100 million tuples and also assume 100 index entries takes up 4KB block.
    - 4GB for an index. Maybe it is too large to store everything in memory.
- Large indexes are stored as a sequential files on the disk
- Searching for an index entry in the large index files will require multiple disk accesses

Multilevel Index

- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
- 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.

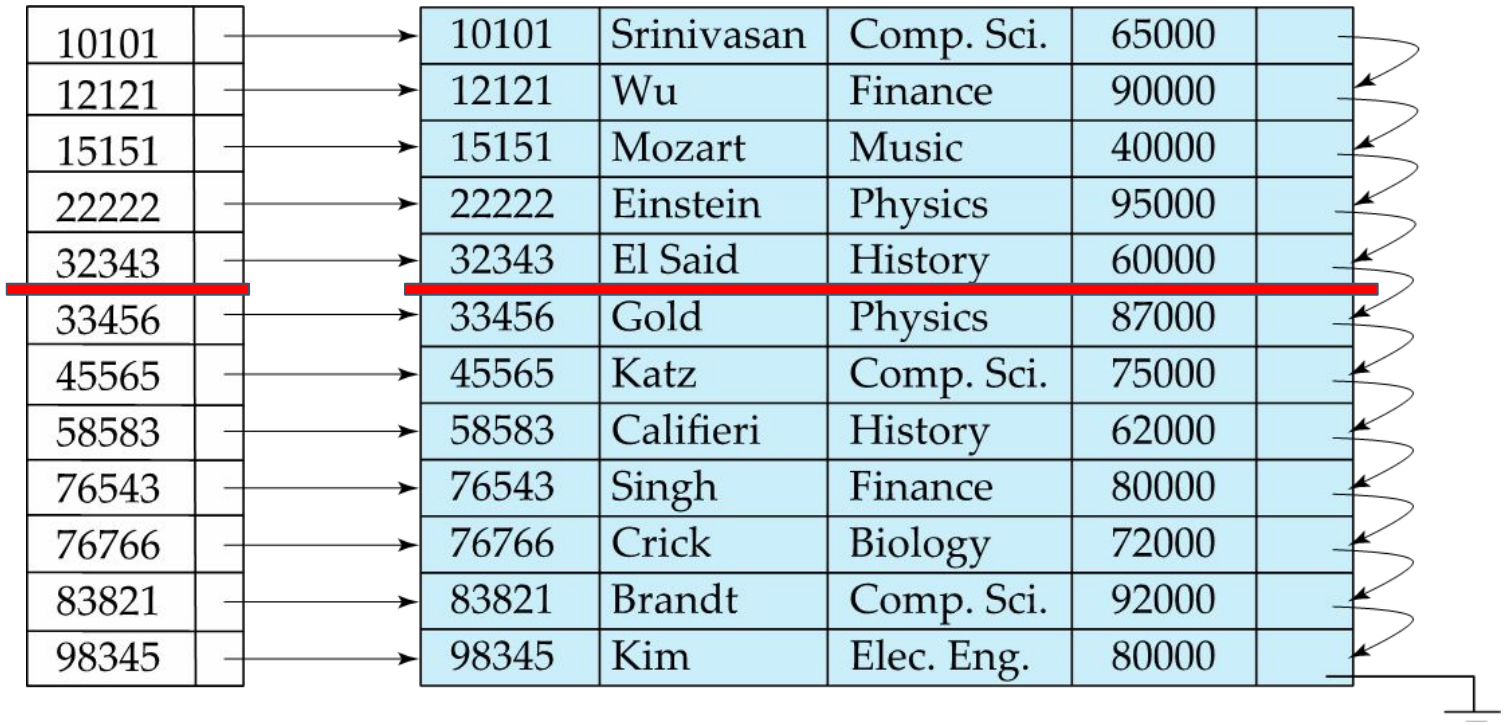


outer index – a sparse index of the basic index

inner index – the basic index file

Index Update: Insertion

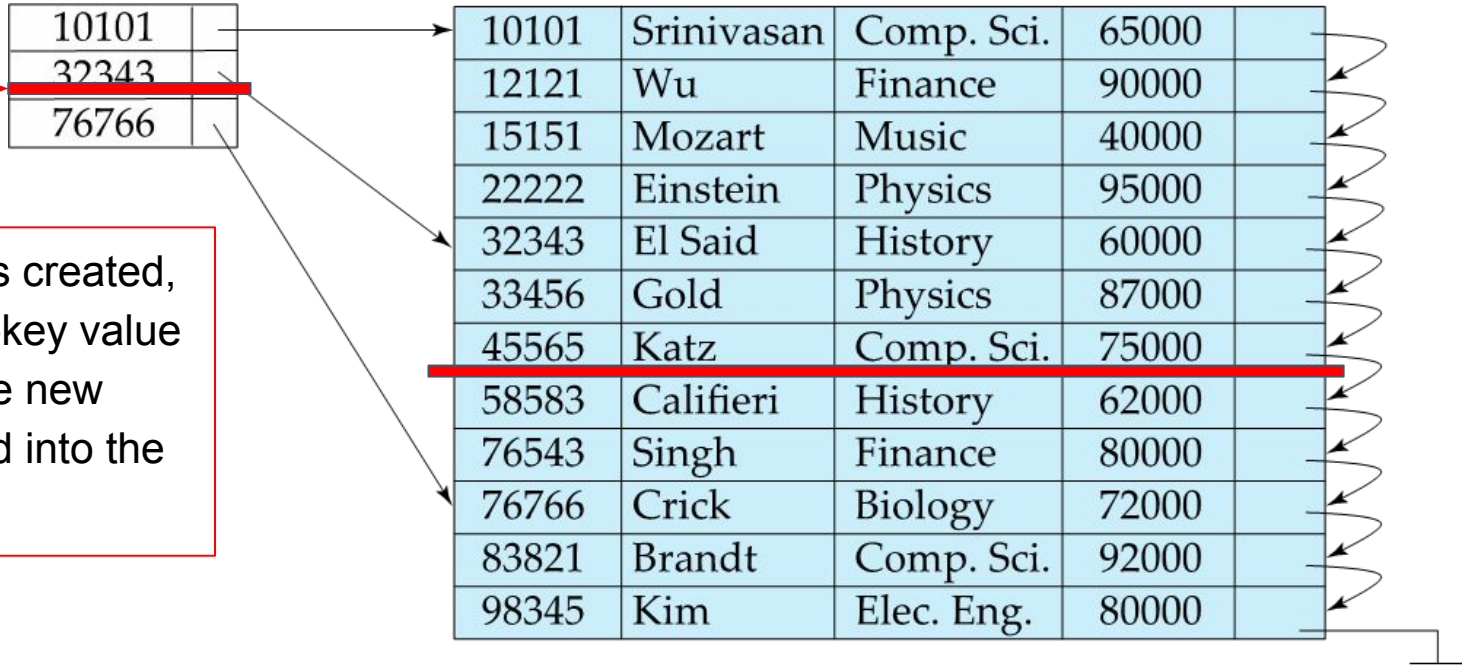
- **Single-level index insertion for Dense Indices:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - If the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required



Insert 33333

Index Update: Insertion

- **Single-level index insertion for Sparce Indices:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

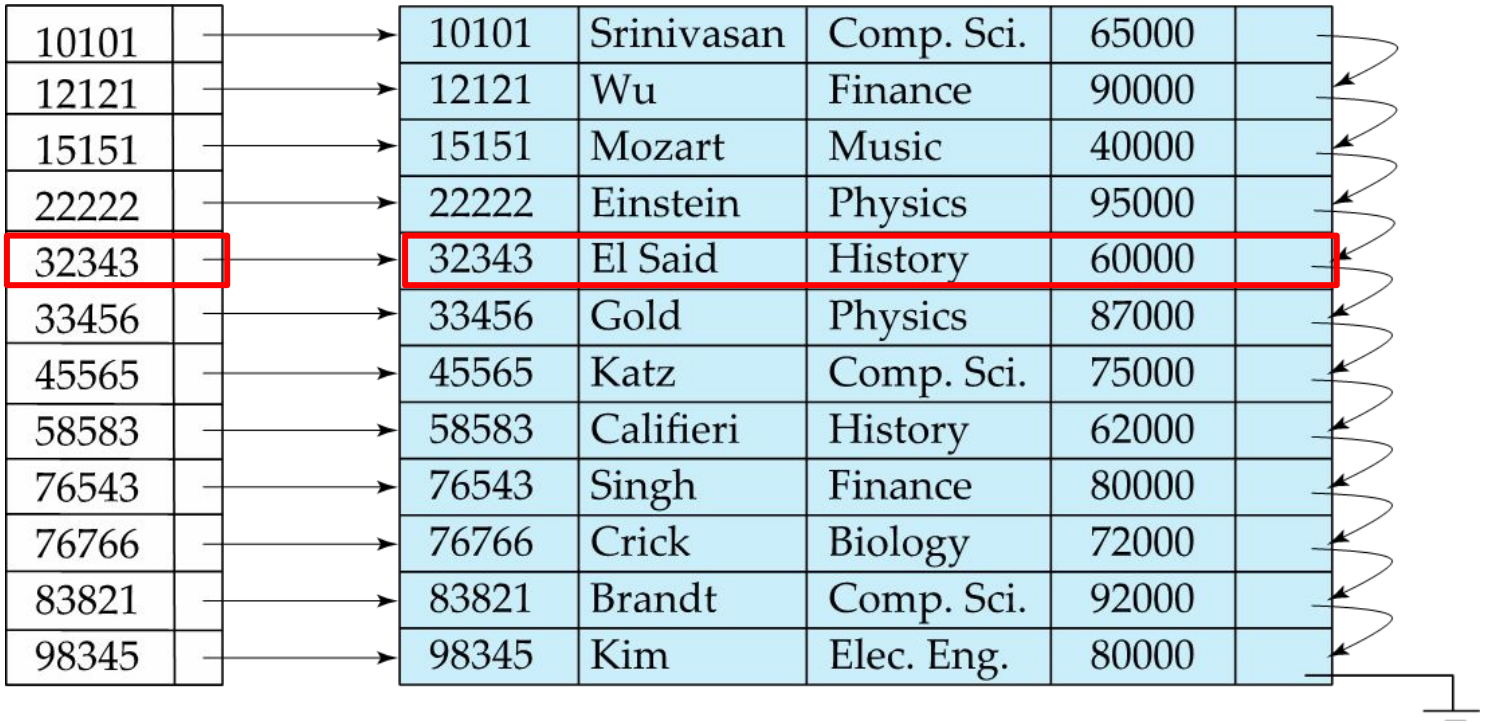


If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Insert 55555

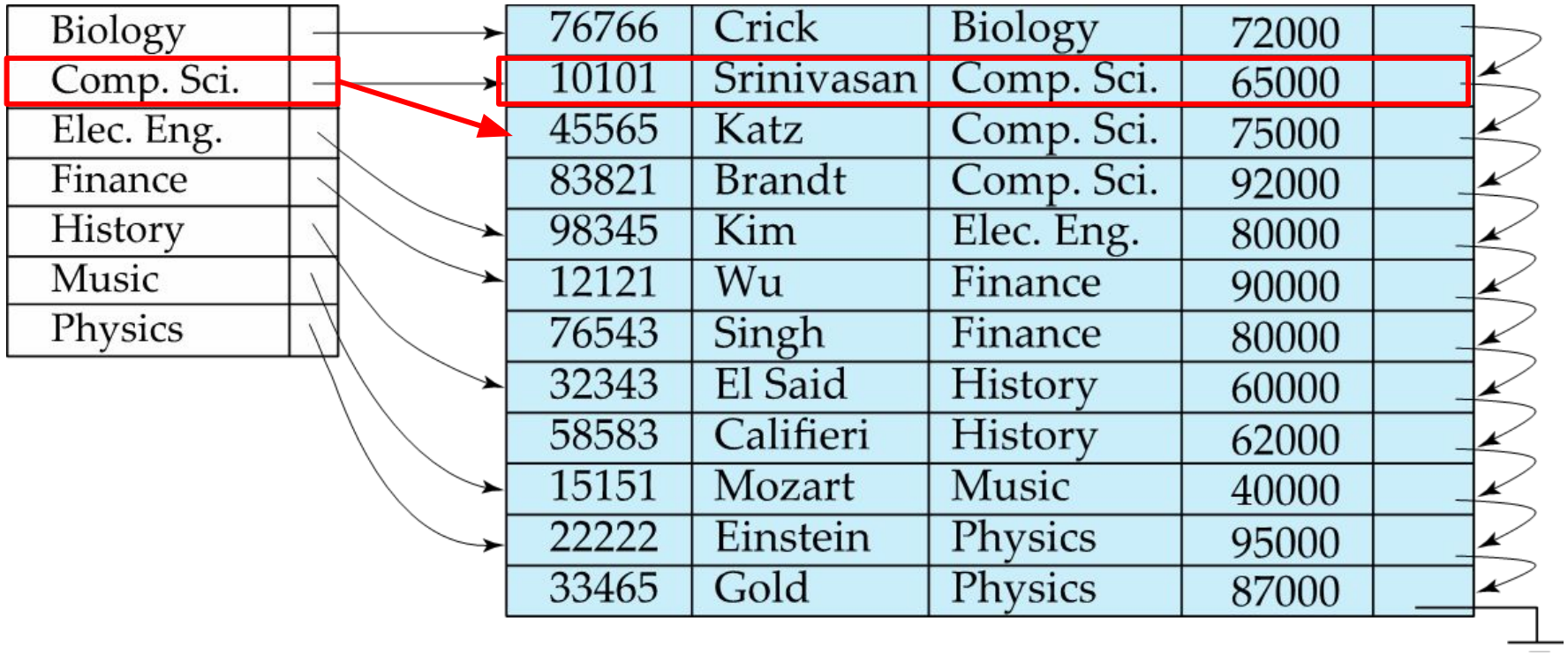
Index Update: Deletion

- **Single-level index deletion for Dense Indices**
  - If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also
    - E.g. If record with ID 32343 is deleted, the index entry will be deleted as well



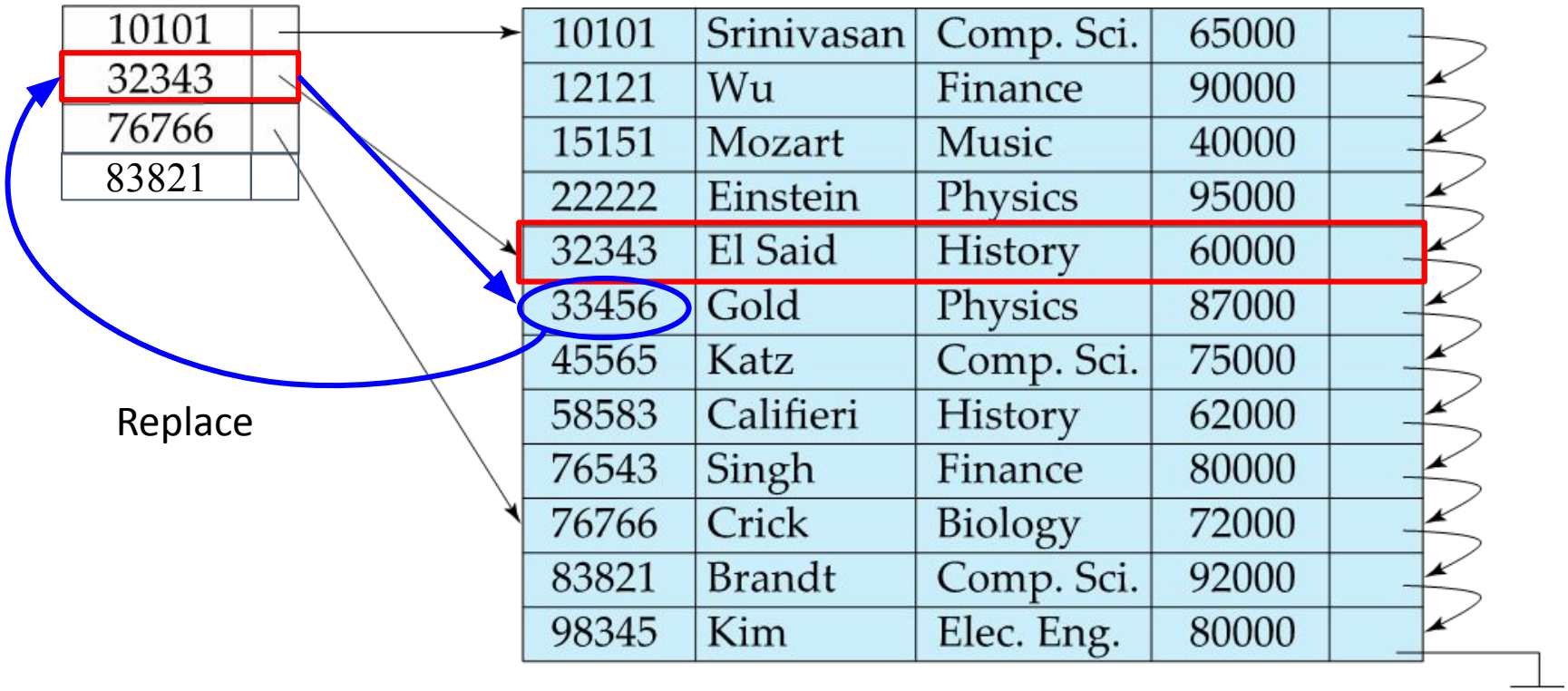
Index Update: Deletion

- **Single-level index deletion for Dense Indices**
  - If deleted record was the first record with the search-key value, the system updates the index entry to point to the next record
    - E.g. if the record with 10101 is deleted, index entry for Comp.Sci. needs to be updated with the new pointer.



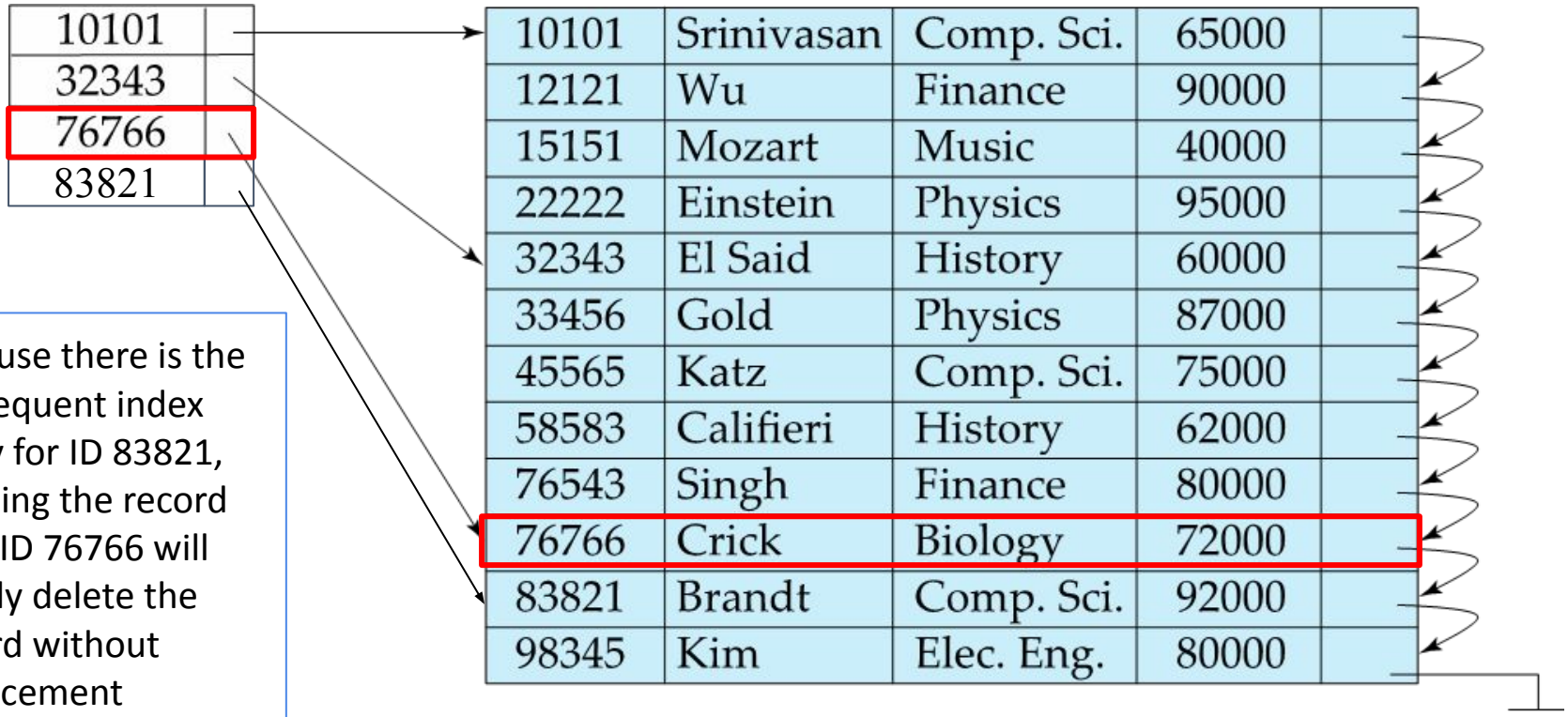
Index Update: Deletion

- **Single-level index deletion for Sparse Indices**
  - If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-ley value



Index Update: Deletion

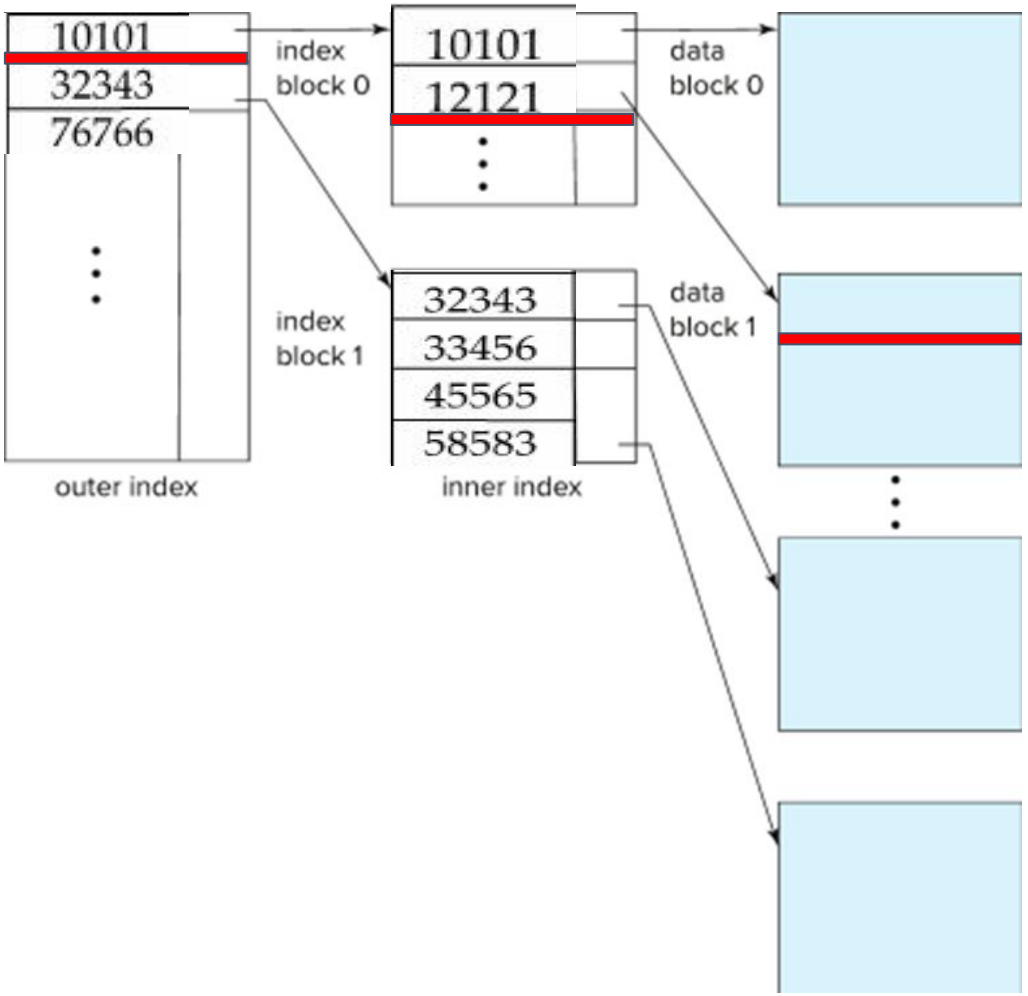
- **Single-level index deletion for Sparse Indices**
  - If the next search-key value of deleted one already has an index entry, the entry is deleted instead of being replaced



Because there is the subsequent index entry for ID 83821, deleting the record with ID 76766 will simply delete the record without replacement

Index Update: Multilevel Indices

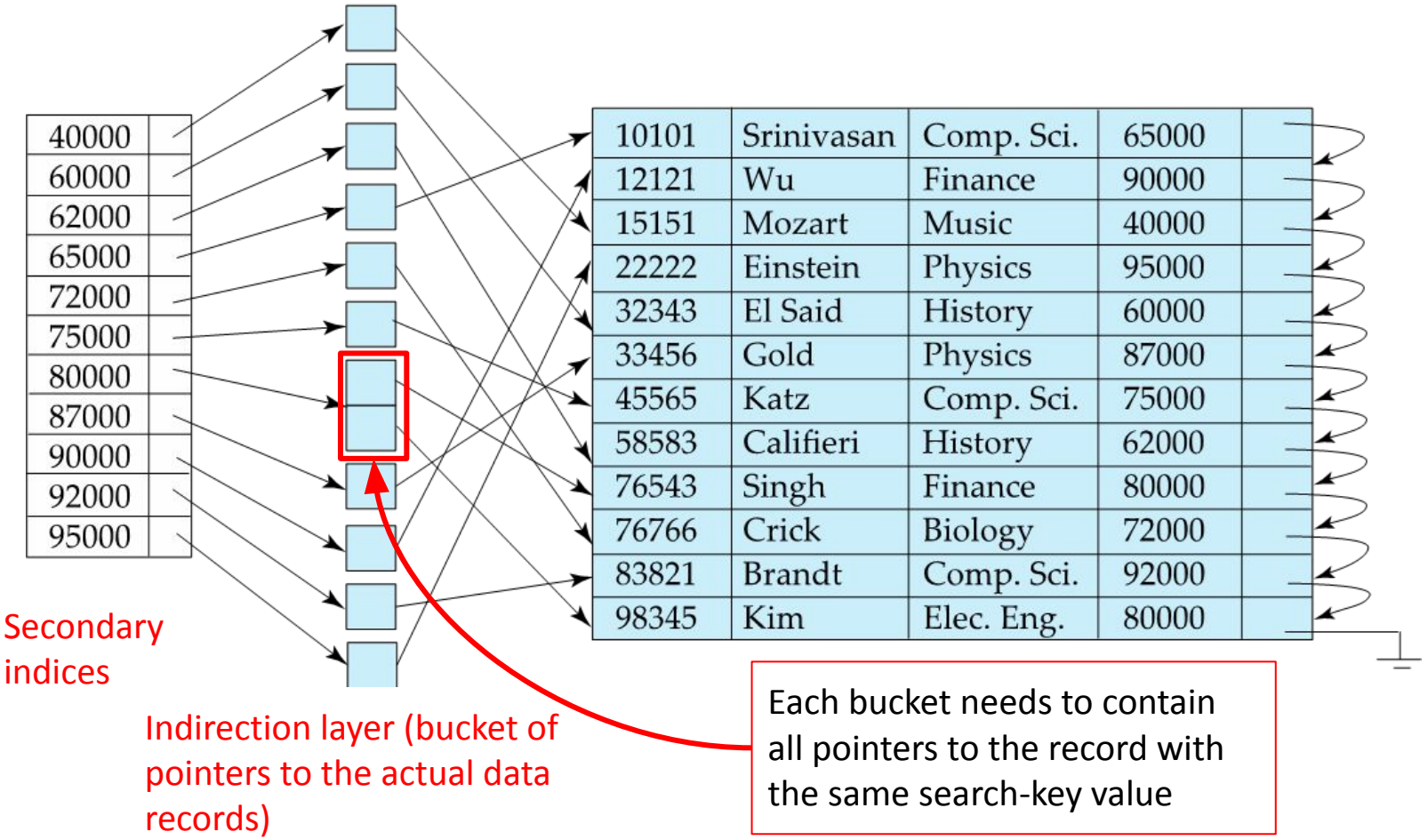
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms



Insert 12222

Secondary Indices Example

- Secondary index (nonclustering index) on salary field of instructor



- Secondary indices have to be dense

Indices on Multiple Keys

- **Composite search key** : A search key containing more than one attribute
  - 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*)

Overview

- Basic Concepts
- Ordered Indices
- Assignments

Assignments

- Reading: Ch14.1, 14.2
- Practice Excercises: 14.1, 14.2

Solutions to the Practice Excercises:
https://www.db-book.com/Practice-Exercises/index-solu.html

The End