Indexing – Part1

Dr. L.M. Jenila Livingston

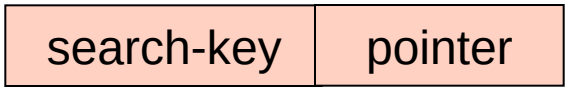
VIT Chennai

Indexing

- Basic Concepts
- Indexing - Ordered Indices
  - Primary Index
    - Dense Index
    - Sparse Index
  - Secondary Index
  - B-Tree Index Files
  - B^+-Tree Index Files

Basic Concepts

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



- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  1. **Ordered indices:** search keys are stored in sorted order, based on the search key
  2. **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

- Access time
- Insertion time
- Deletion time
- Space overhead

Index Evaluation Metrics

- In an Online Transaction Processing (OLTP) environment - Insertion, deletion and update time are important.
- In a Decision Support Systems (DSS) environment - access time is important:
  - Point Queries - Records with a specified value in an attribute.
  - Range Queries – Records with an attribute value in a specified range.
- In either case, space used is also important.

Ordered Indices

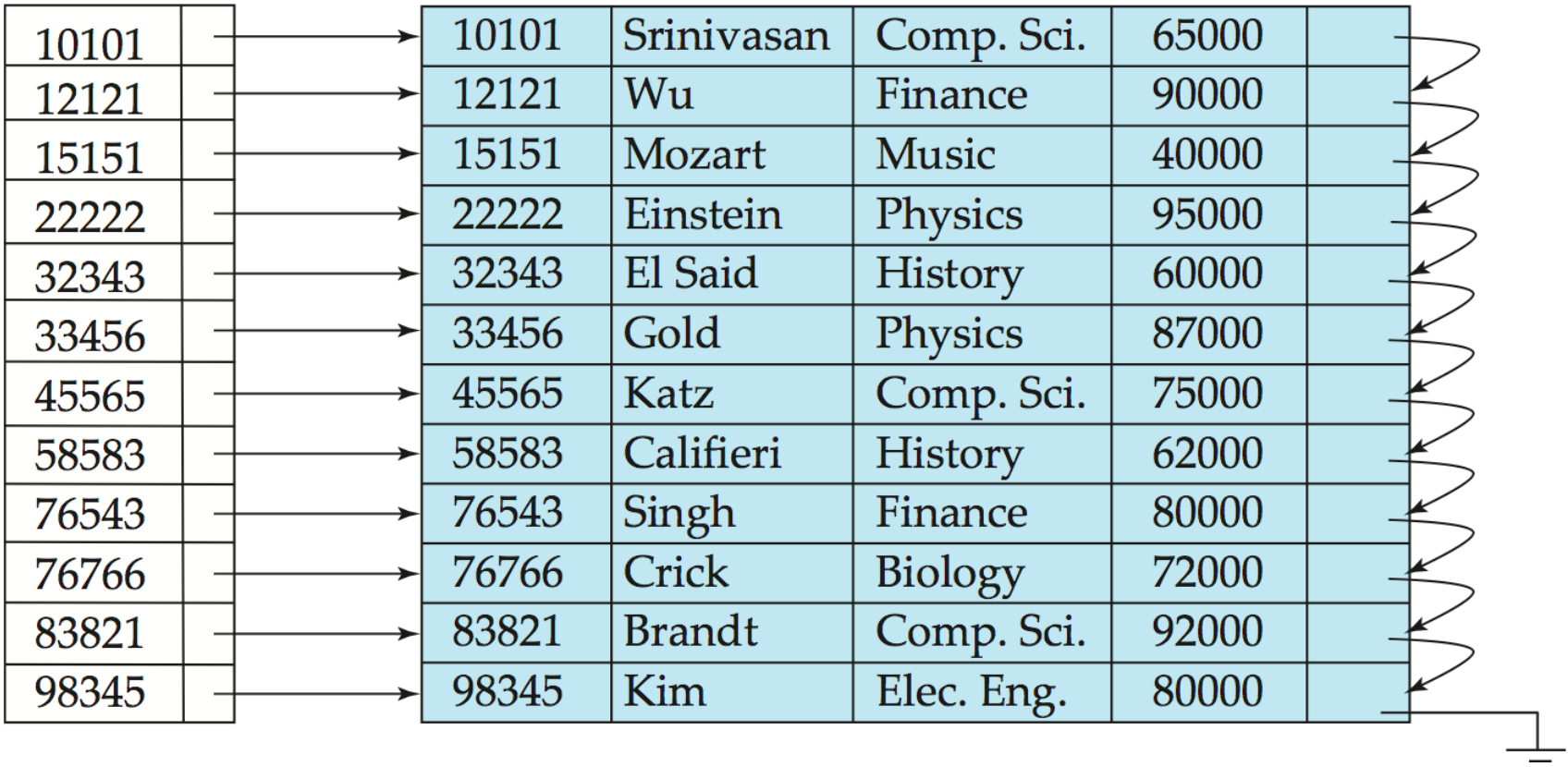
- **Primary index:** An index whose search key specifies the **sequential order** of the data file is a *primary index*.
  - Also called *clustering* or *clustered index.*
  - Search key of a primary index is frequently the primary key.
  - An ordered sequential file with a primary index is an *index-sequential file*.
- **Secondary index**: An index whose search key **does not specify the sequential order** of the data file is a *secondary index*.
  - Also called a *non-clustering* or non-clustered index**.**

Primary index

- Dense Index
- Sparse Index
- Multi-Level Index

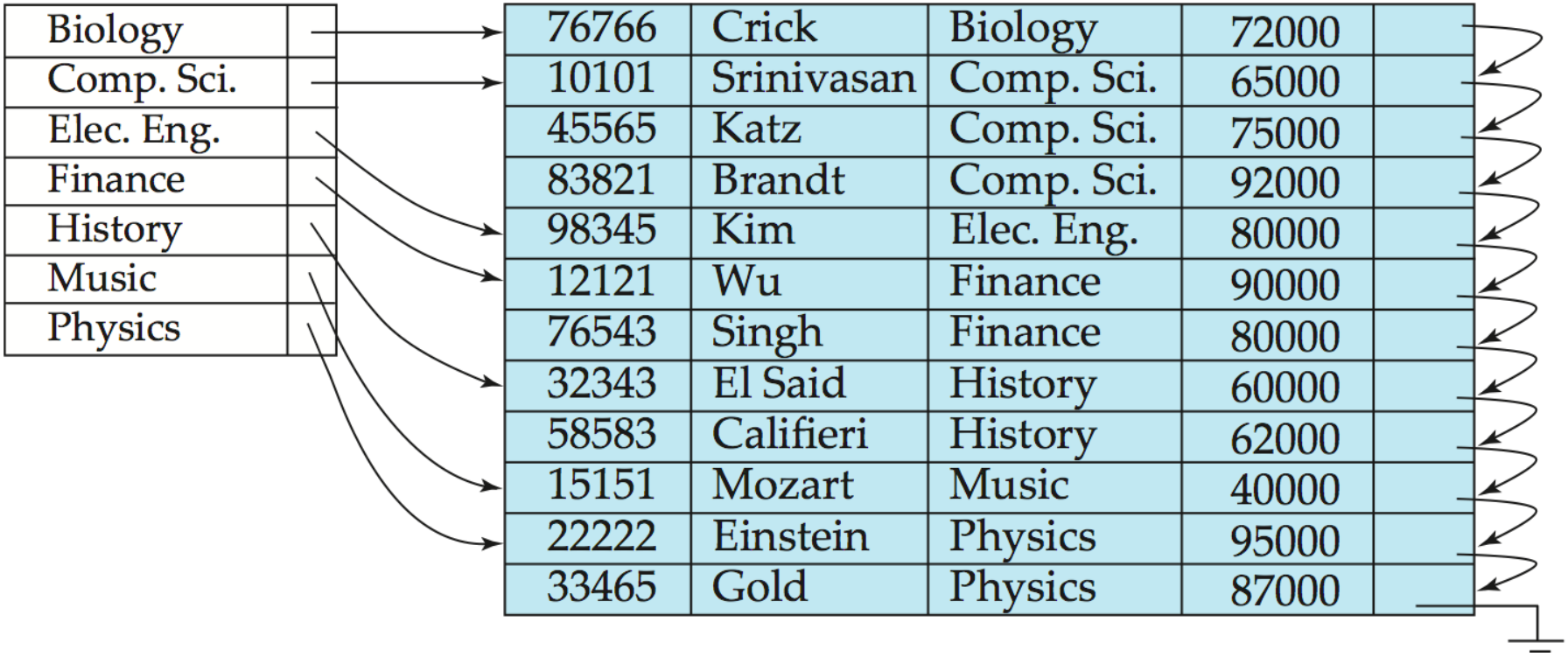
Dense Index Files

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



Dense Index Files (Cont.)

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

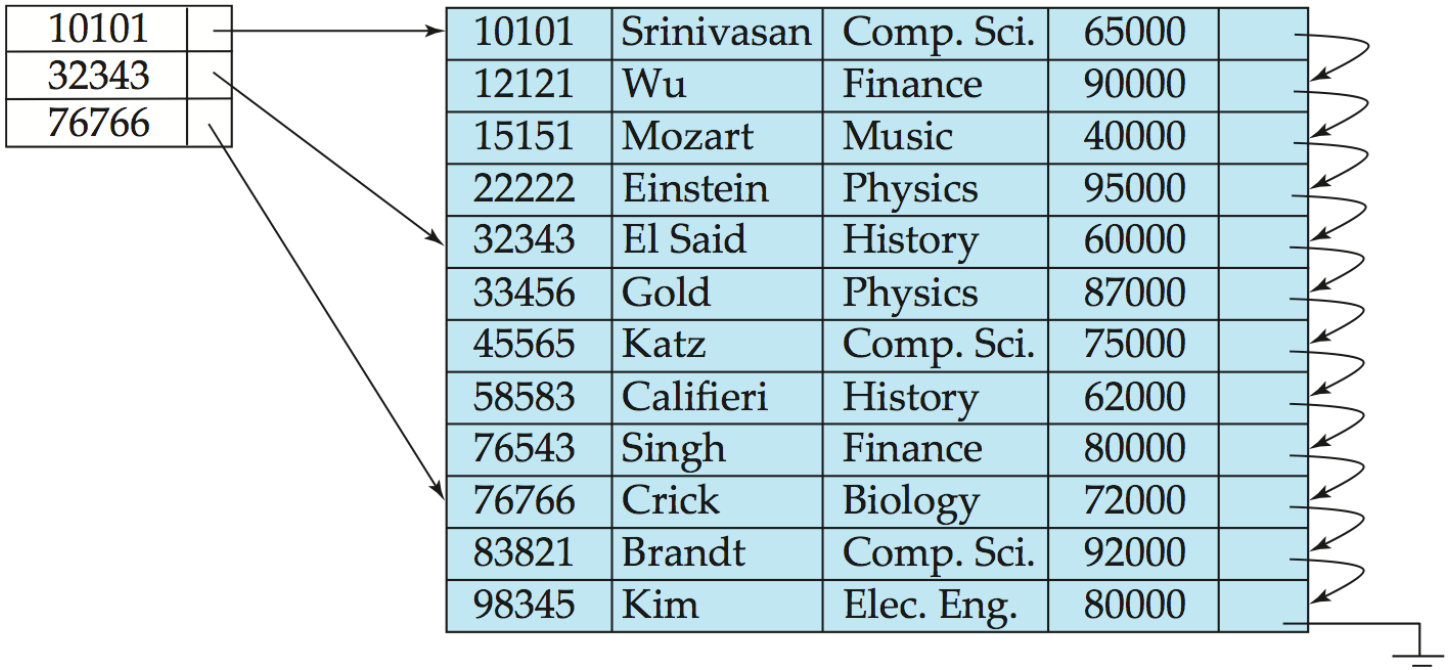


Dense Index Files, Cont.

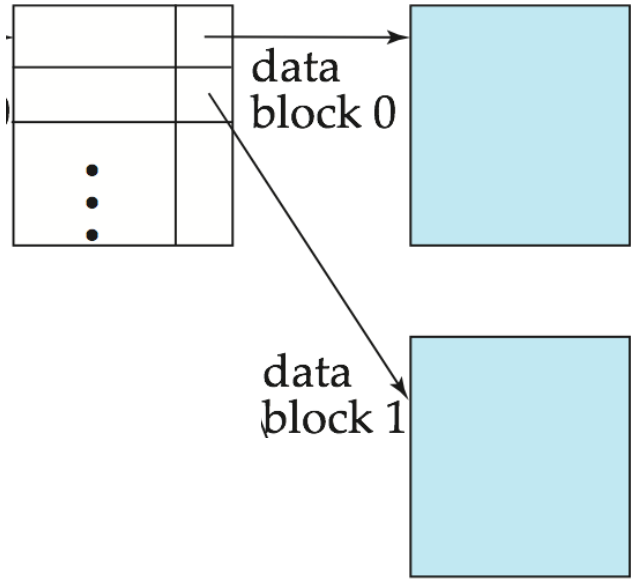
- To locate the record(s) with search-key value *K*:
  - Find index record with search-key value *K.*
  - Follow pointer from the index record to the data record(s).
- To delete a record:
  - Locate the record in the data file, perhaps using the above procedure.
  - Delete the record from the data file.
  - If the deleted record was the only one with that search-key value, then delete the search-key from the index (similar to data record deletion)
- To insert a record:
  - Perform an index lookup using the records' search-key value.
  - If the search-key value appears in the index, following the pointer to the data file and insert the record.
  - If the search-key value does not appear in the index:
    - insert the search key into the index file
    - insert the record into the data file in an appropriate place
    - assign a pointer to the data record from the index record.

Sparse Index Files

- An index that contains index records but only for some search-key values in the data file is a *sparse index*.
- Typically one index entry for each data file block.



Sparse Index Files (Cont.)



Sparse Index Files, Cont.

- Advantages (relative to dense indices):
  - Require less space
  - Less maintenance for insertions and deletions
- Disadvantages:
  - Slower for locating records, especially if there is more than one block per index entry

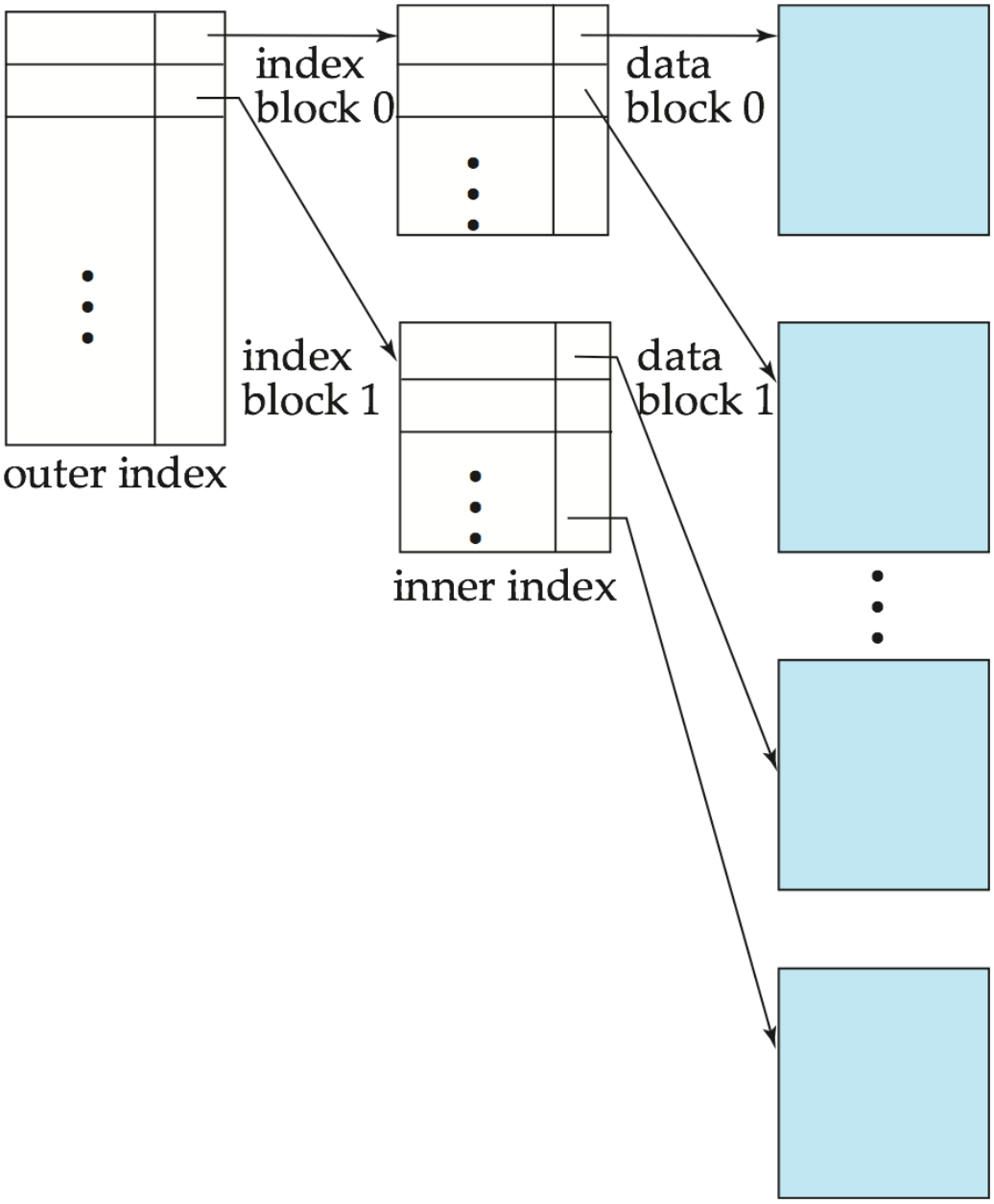
Sparse Index Files, Cont.

- To locate a record with search-key value *K*:
  - Find the index record with largest search-key value <= *K.*
  - Search file sequentially from the record to which the index record points.
- To delete a record:
  - Locate the record in the data file, perhaps using the above procedure.
  - Delete the record from the data file.
  - If the deleted record was the only record with its search-key value, and if an entry for the search key exists in the index, then replace the index entry with the next search-key value in the data file (in search-key order). If the next search-key value already has an index entry, the index entry is simply deleted.
- To insert a record: (assume the index stores an entry for each data block)
  - Perform an index lookup using the records' search-key value.
  - If the index entry points to a block with free space, then simply insert the record in that block, in sorted order.
  - If the index entry points to a full block, then allocate a new block and insert the first search-key value appearing in the new block into the index

Multilevel Index

- In order to improve performance, an attempt is frequently made to store, i.e., pin, all index blocks in memory.
- Unfortunately, sometimes an index is too big to fit into memory.
- In such a case, the index can be treated as a sequential file on disk and a sparse index is built on it:
  - outer index – a sparse index
  - inner index – sparse or dense index
- If the outer index is still too large to fit in main memory, yet another level of index can be created, and so on.

Multilevel Index (Cont.)



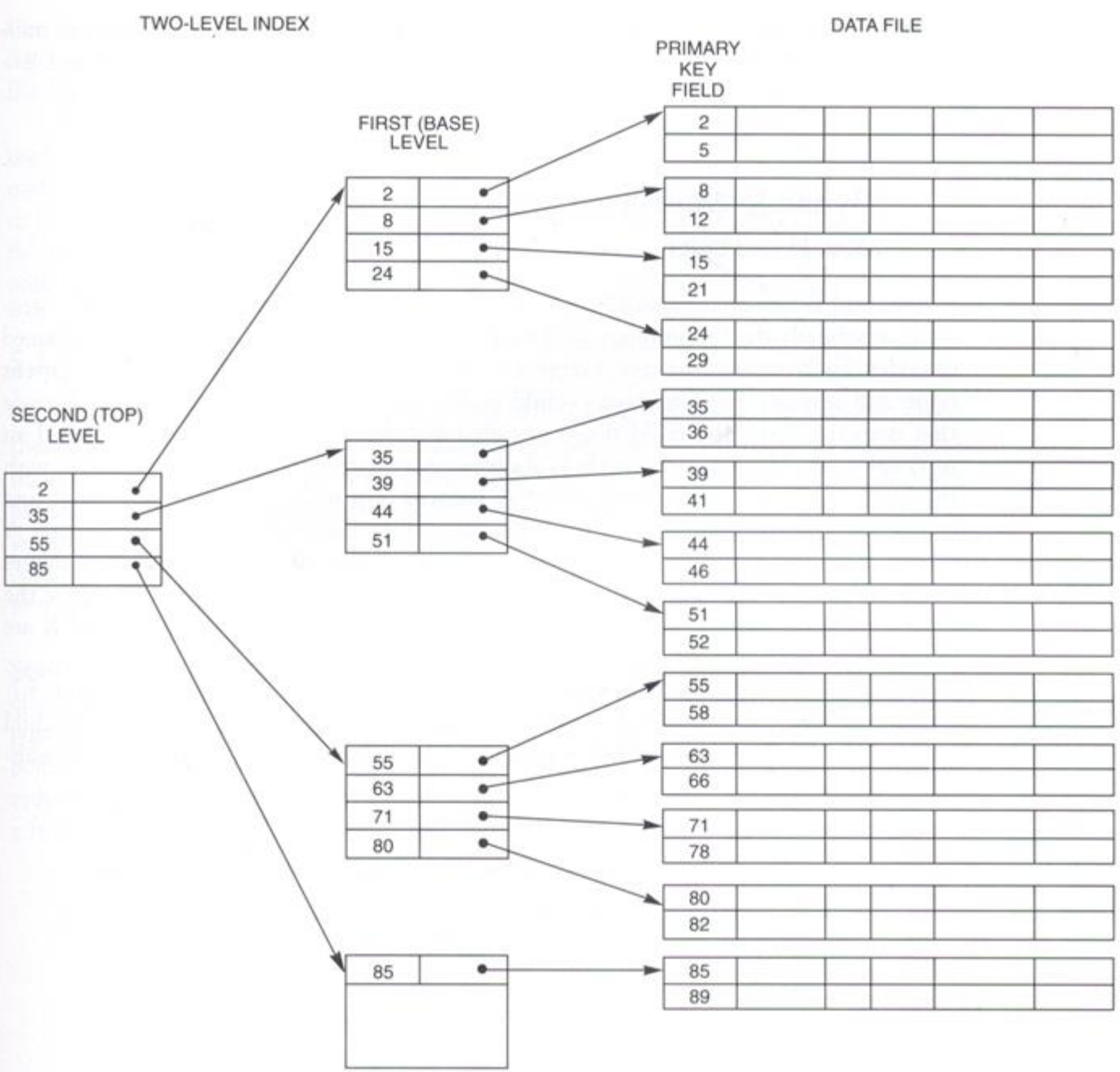


Figure 5.6 A two-level primary index.

Multilevel Index, Cont.

- Indices at all levels might require updating upon insertion or deletion.
- Multilevel insertion, deletion and lookup algorithms are simple extensions of the single-level algorithms.

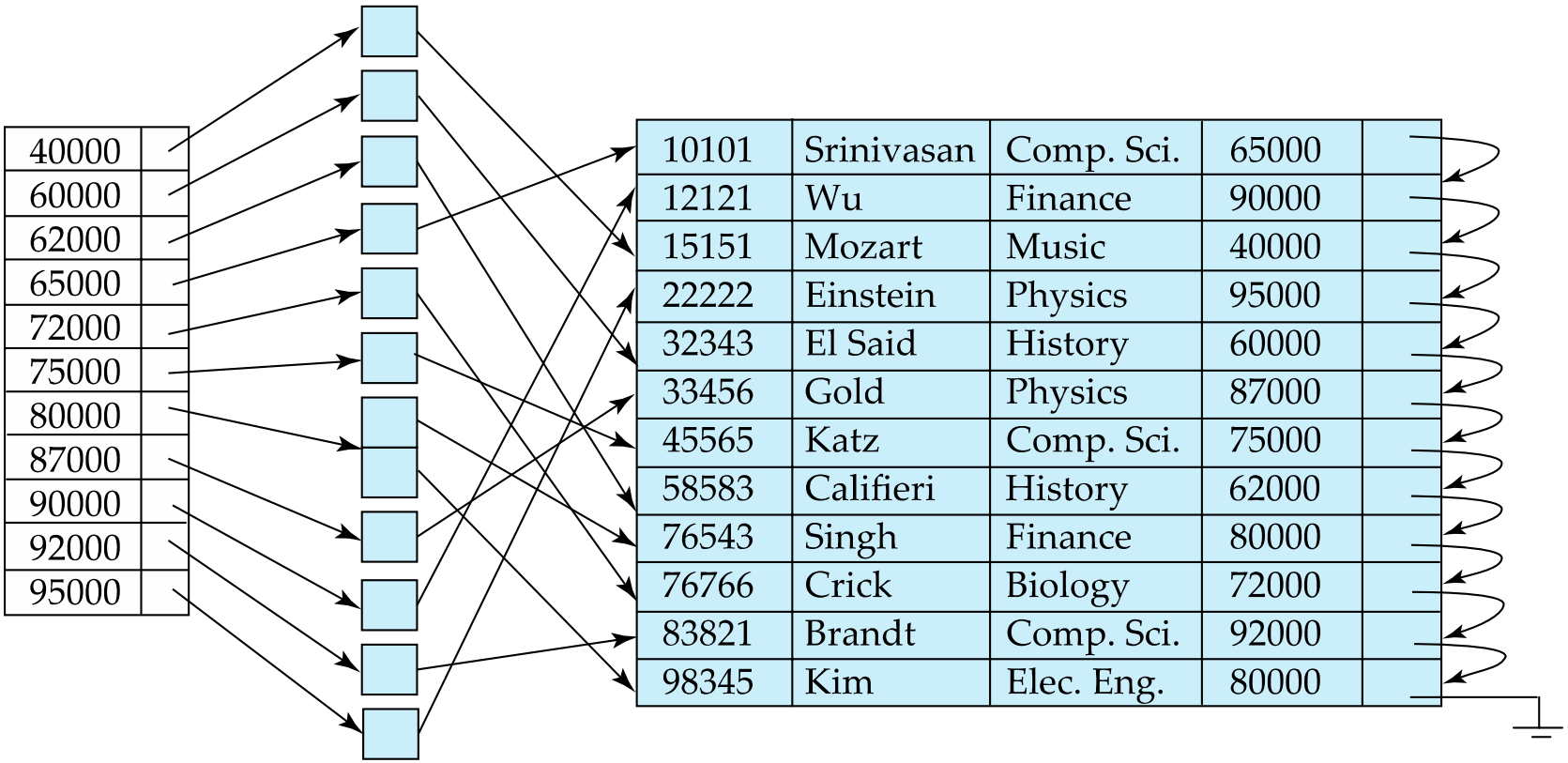
Secondary Indices

- So far, our consideration of dense and sparse indices has only been in the context of primary indices.
- Recall that an index whose search key does not specify the sequential order of the data file is called a *secondary index*.
- A secondary index is used when a table is searched using a search key other than the one on which the table is sorted.
  - Suppose *account* is sorted by account number, but searches are based on branch, or searching for a range of balances.
  - Suppose payment is sorted by *loan#* and *payment#*, but searches are based on *id#*

Secondary Indices

- In a secondary index, each index entry will point to either a:
  - Single record containing the search key value (candidate key).
  - Bucket that contains pointers to all records with that search-key value (non-candidate key).
- All previous algorithms and data structures can be modified to apply to secondary indices.

Secondary Indices Example



Secondary index on *salary* field of *instructor*

- 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

Secondary 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

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

Index Classification

- In summary, the indices we have considered so far are either:
  - Dense, or
  - Sparse
- In addition, an index may be either:
  - Primary, or
  - Secondary
- And the search key the index is built on may be either a:
  - Candidate key
  - Non-candidate key
- Note, that the book claims a secondary index must be dense; why?