UMBC

AN HONORS UNIVERSITY IN MARYLAND

CMSC 461, Database Management Systems
Spring 2018

Lecture 16 – Chapter 11 Indexing and Hashing Part 1

These slides are based on "Database System Concepts" 6th edition book (whereas some quotes and figures are used from the book) and are a modified version of the slides which accompany the book (http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html), in addition to the 2009/2012 CMSC 461 slides by Dr. Kalpakis

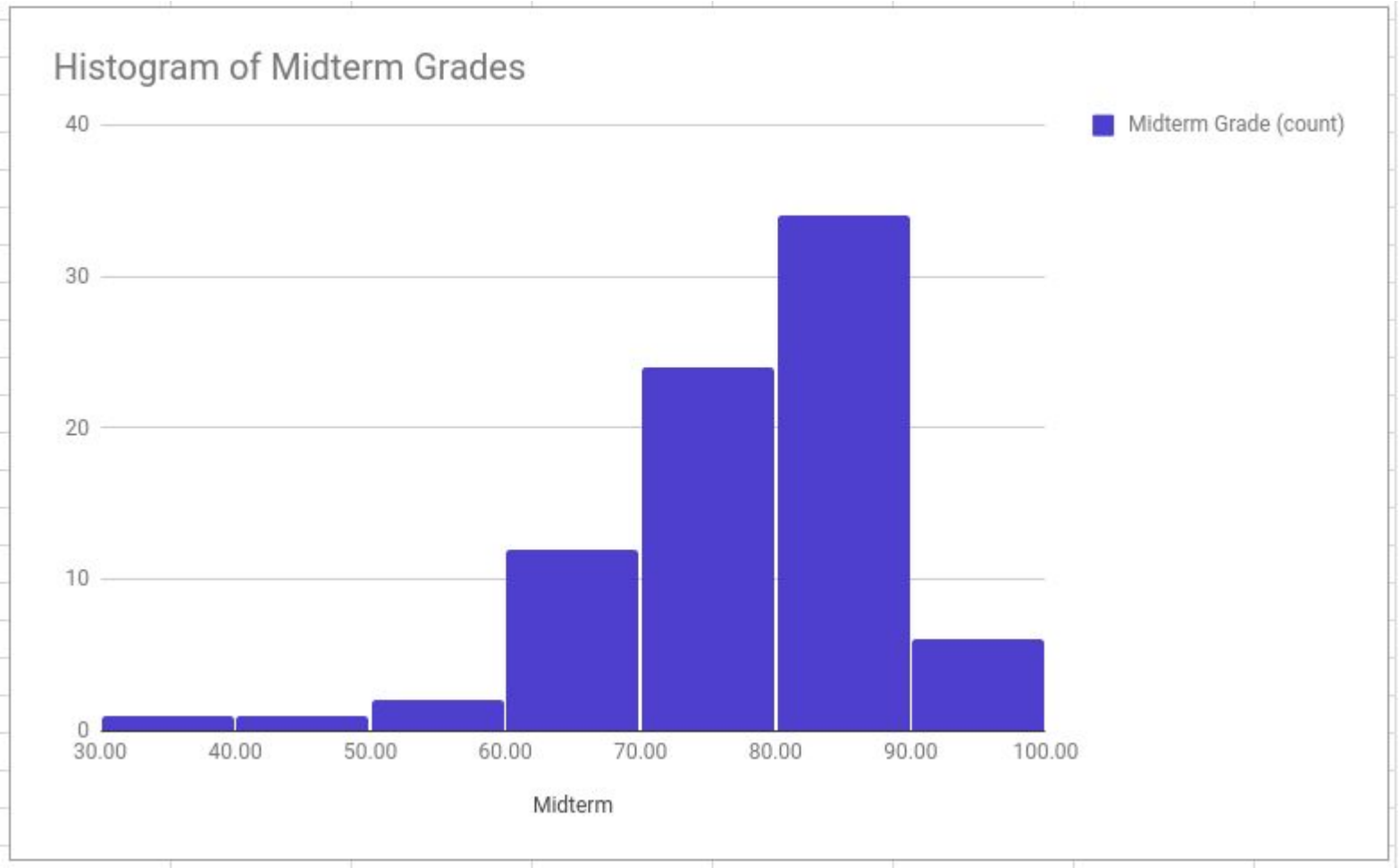
Dr. Jennifer Sleeman

https://www.csee.umbc.edu/~jsleem1/courses/461/spr18

Logistics

- Project Phase 2 due
- Homework #4 will be available this evening

Logistics


Histogram of Midterm Grades
Midterm Grade (count)
40
30
20
10
0
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00
Midterm

Lecture Outline

- Summary of Storage and File Organization
- Indexing

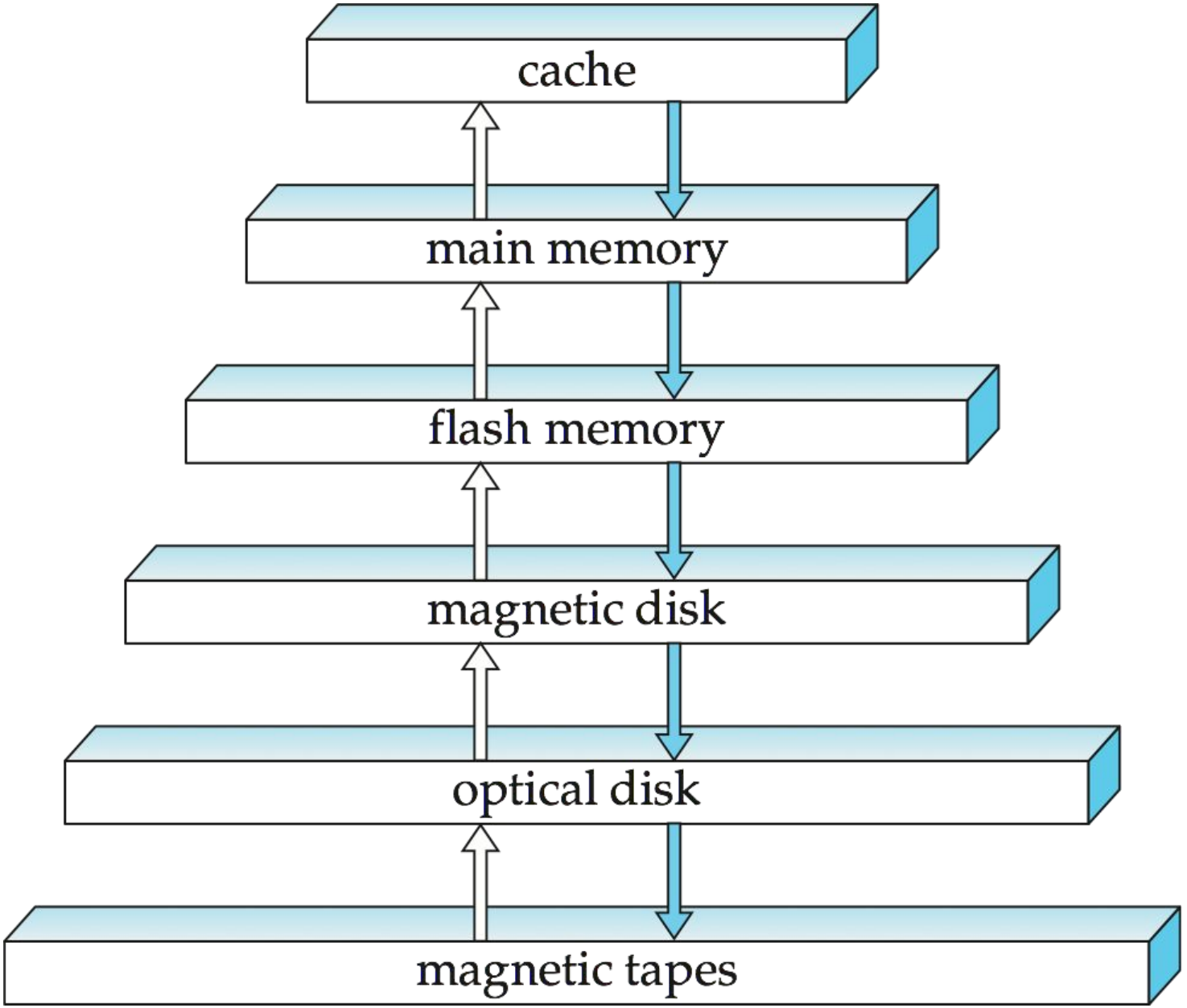
Lecture Outline

- ***Summary of Storage and File Organization***
- Indexing

Summary of Storage and File Organization

- DBMS typically stores data on disk
- Try to minimize overhead of moving between disk and memory
  - Performance measures
  - Optimizations for block access
  - RAID – High capacity and reliability

Storage Hierarchy



Based on and image from "Database System Concepts" book and slides, 6th edition

Summary of Storage and File Organization

- The database is stored as a collection of *files*
- Each file is a sequence of *records*
- A record is a sequence of fields
- Records are mapped onto disk blocks
  - Each file logically partitioned into blocks

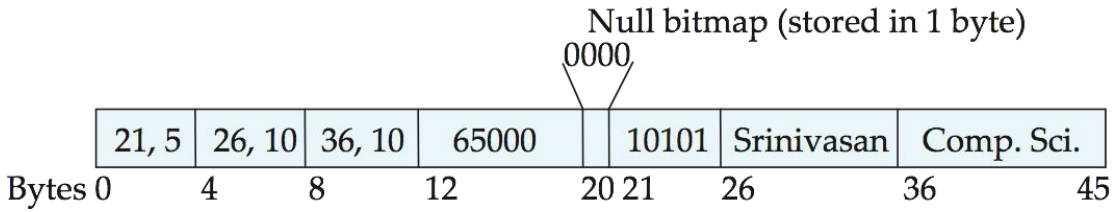
Summary of Storage and File Organization

- Blocks are the units of storage allocation and data transfer
  - Usually 4 to 8 kilobytes
- A block can contain many records
- The physical data organization determines how many records contained in a block

Summary of Storage and File Organization

- Fixed length records
  - Simple access
  - Records could cross blocks
  - Free Lists
- Variable length records
  - Storage of multiple record types in a file
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models)

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Based on and image from "Database System Concepts" book and slides, 6th edition

Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record

Organization of Records in Files

- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Lecture Outline

- Summary of Storage and File Organization
- ***Indexing***

Indexing

```
select * from section;
```

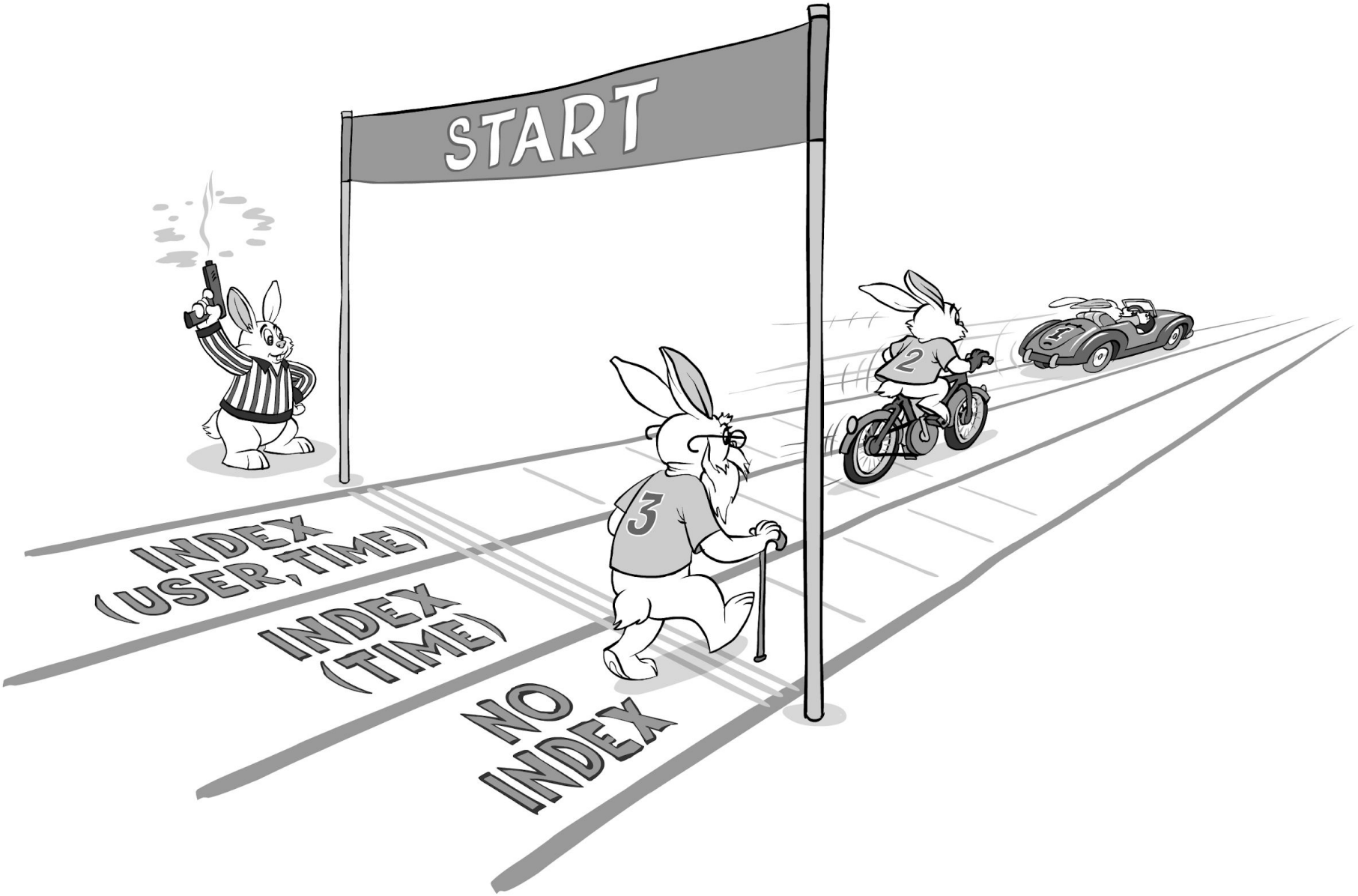
```
+-----------+--------+----------+------+----------+-------------+--------------+
| course_id | sec_id | semester | year | building | room_number | time_slot_id |
+-----------+--------+----------+------+----------+-------------+--------------+
| BIO-101   | 1      | Summer   | 2009 | Painter  | 514         | B            |
| BIO-301   | 1      | Summer   | 2010 | Painter  | 514         | A            |
| CS-101    | 1      | Fall     | 2009 | Packard  | 101         | H            |
| CS-101    | 1      | Spring   | 2010 | Packard  | 101         | F            |
| CS-190    | 1      | Spring   | 2009 | Taylor   | 3128        | E            |
| CS-190    | 2      | Spring   | 2009 | Taylor   | 3128        | A            |
| CS-315    | 1      | Spring   | 2010 | Watson   | 120         | D            |
| CS-319    | 1      | Spring   | 2010 | Watson   | 100         | B            |
| CS-319    | 2      | Spring   | 2010 | Taylor   | 3128        | C            |
| CS-347    | 1      | Fall     | 2009 | Taylor   | 3128        | A            |
| EE-181    | 1      | Spring   | 2009 | Taylor   | 3128        | C            |
| FIN-201   | 1      | Spring   | 2010 | Packard  | 101         | B            |
| HIS-351   | 1      | Spring   | 2010 | Painter  | 514         | C            |
| MU-199    | 1      | Spring   | 2010 | Packard  | 101         | D            |
| PHY-101   | 1      | Fall     | 2009 | Watson   | 100         | A            |
+-----------+--------+----------+------+----------+-------------+--------------+
```

Indexing

```
select * from section where course_id = 'BIO-101' or
course_id = 'BIO-301';
```

```
+-----------+--------+----------+------+----------+-------------+-------------+
| course_id | sec_id | semester | year | building | room_number | time_slot_id |
+-----------+--------+----------+------+----------+-------------+-------------+
| BIO-101   | 1      | Summer   | 2009 | Painter  | 514         | B           |
| BIO-301   | 1      | Summer   | 2010 | Painter  | 514         | A           |
+-----------+--------+----------+------+----------+-------------+-------------+
```

Indexing



Indexing

Index

Based on and image from "Database System Concepts" book and slides, 6th edition

Indexing

```
select * from section where course_id = 'BIO-101' or
course_id = 'BIO-301';
```

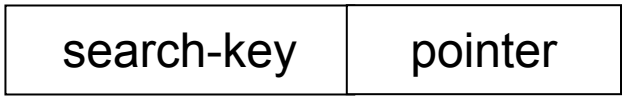
```
+-----------+--------+----------+------+----------+-------------+-------------+
| course_id | sec_id | semester | year | building | room_number | time_slot_id |
+-----------+--------+----------+------+----------+-------------+-------------+
| BIO-101   | 1      | Summer   | 2009 | Painter  | 514         | B           |
| BIO-301   | 1      | Summer   | 2010 | Painter  | 514         | A           |
+-----------+--------+----------+------+----------+-------------+-------------+
```

Indexing

- Queries accessing small portion of data
  - Don't read more than needed
  - Locate records directly
- Indexing – think of an index in a book
  - Sorted
  - Smaller than the data itself

Basic Concepts

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



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”.

Index Evaluation Metrics

- Different techniques for different applications
- In general evaluate using the following:
  - Access types
    - supported efficiently
    - Find records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
  - Access time
    - Time it takes to find item

Index Evaluation Metrics

- Insertion time
  - Time it takes to insert new item
- Deletion time
  - Time it takes to delete item
- Space overhead
  - Additional space occupied by index structure
  - Good to sacrifice space for better performance

Ordered Indices

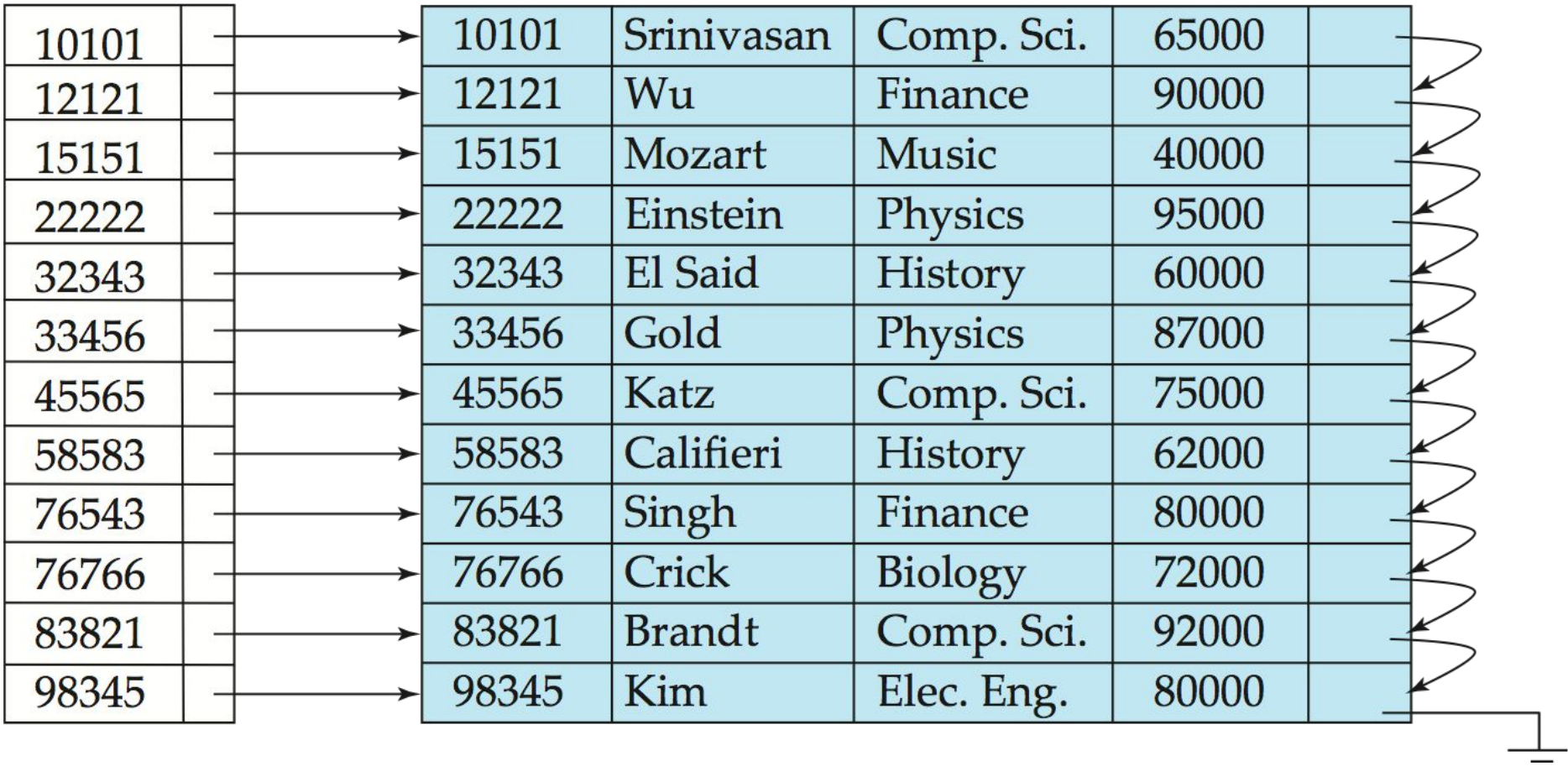
- In an **ordered index**, index entries are stored sorted on the search key value
  - Similar to catalog in library
  - Search keys in sorted order
  - Search key to records
  - Files can have many search keys
- **Primary index** - in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.

Ordered Indices

- **Secondary index** - an index whose search key specifies an order different from the sequential order of the file
  - Also called non-clustering index
- Index-sequential file: ordered sequential file with a primary index.

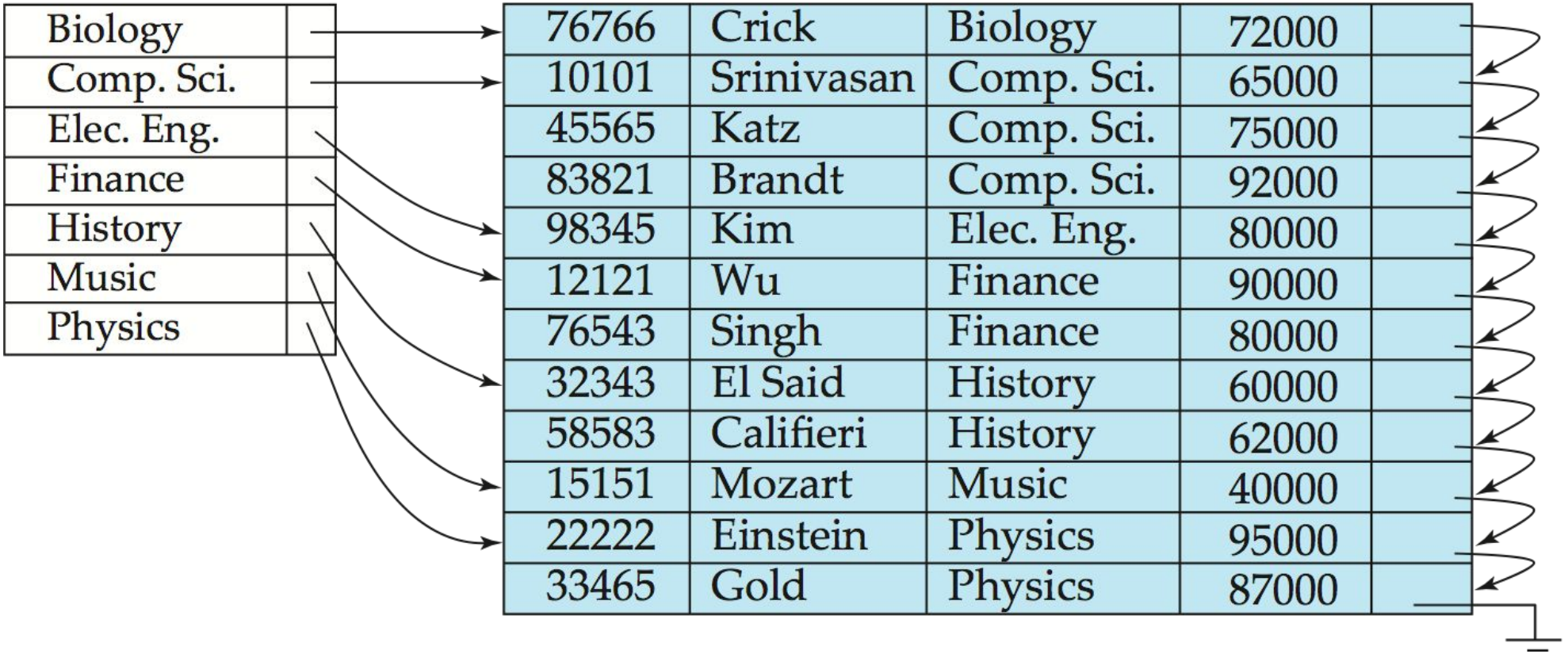
Dense Index Files

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



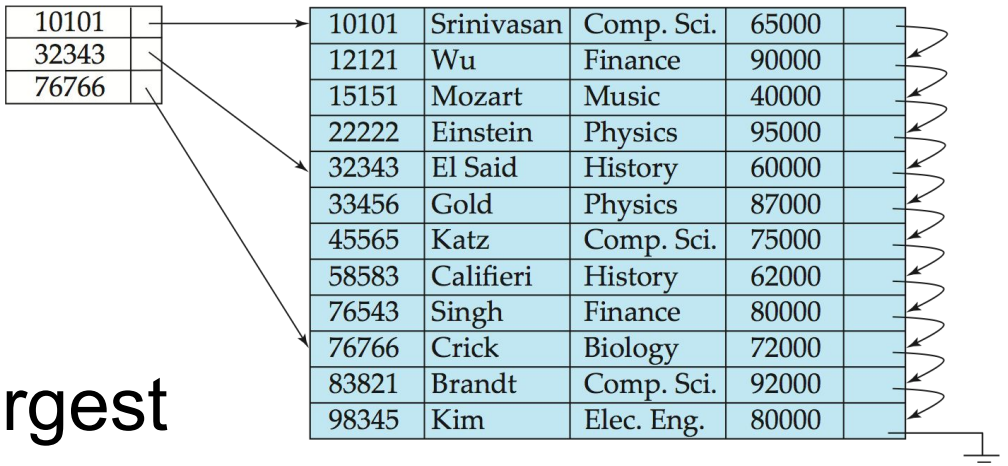
Dense Index Files

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

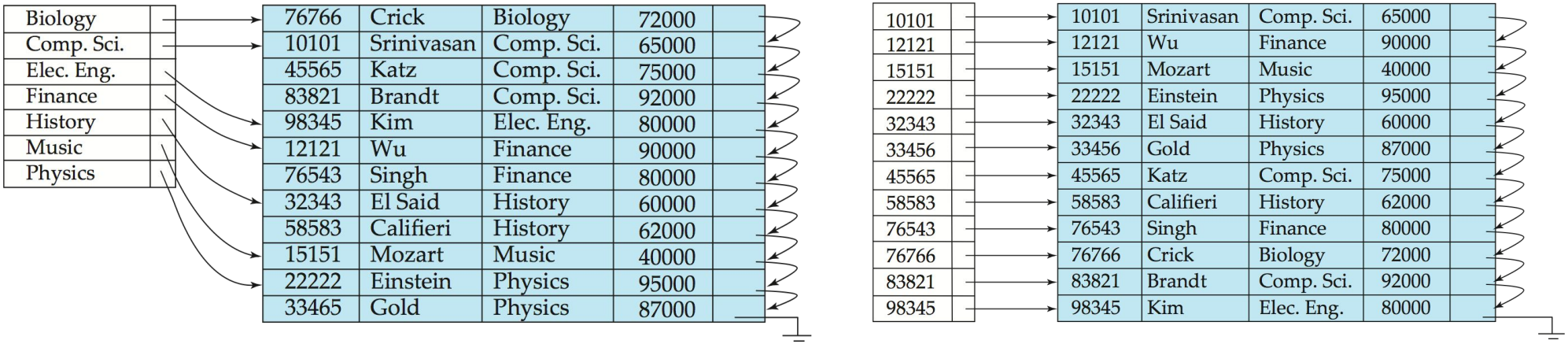


Sparse Index Files

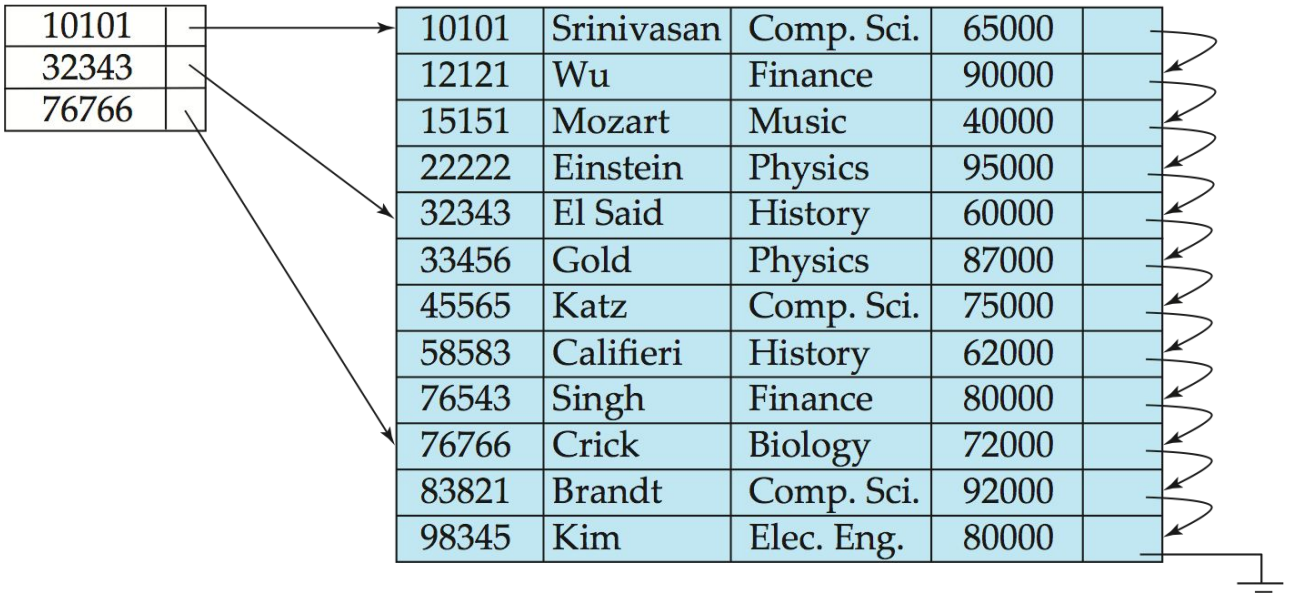
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record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



Dense/Sparse Index Files



Dense Index File

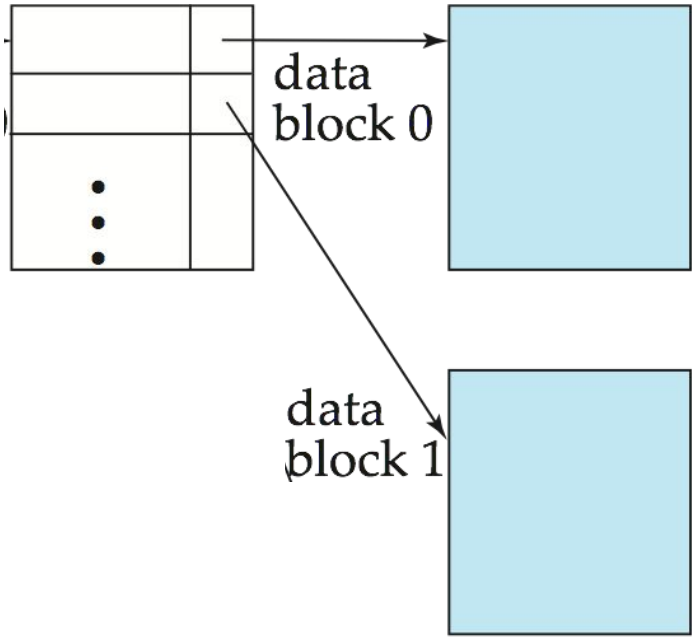


Sparse Index File

Based on and image from "Database System Concepts" book and slides, 6th edition

Sparse Index Files

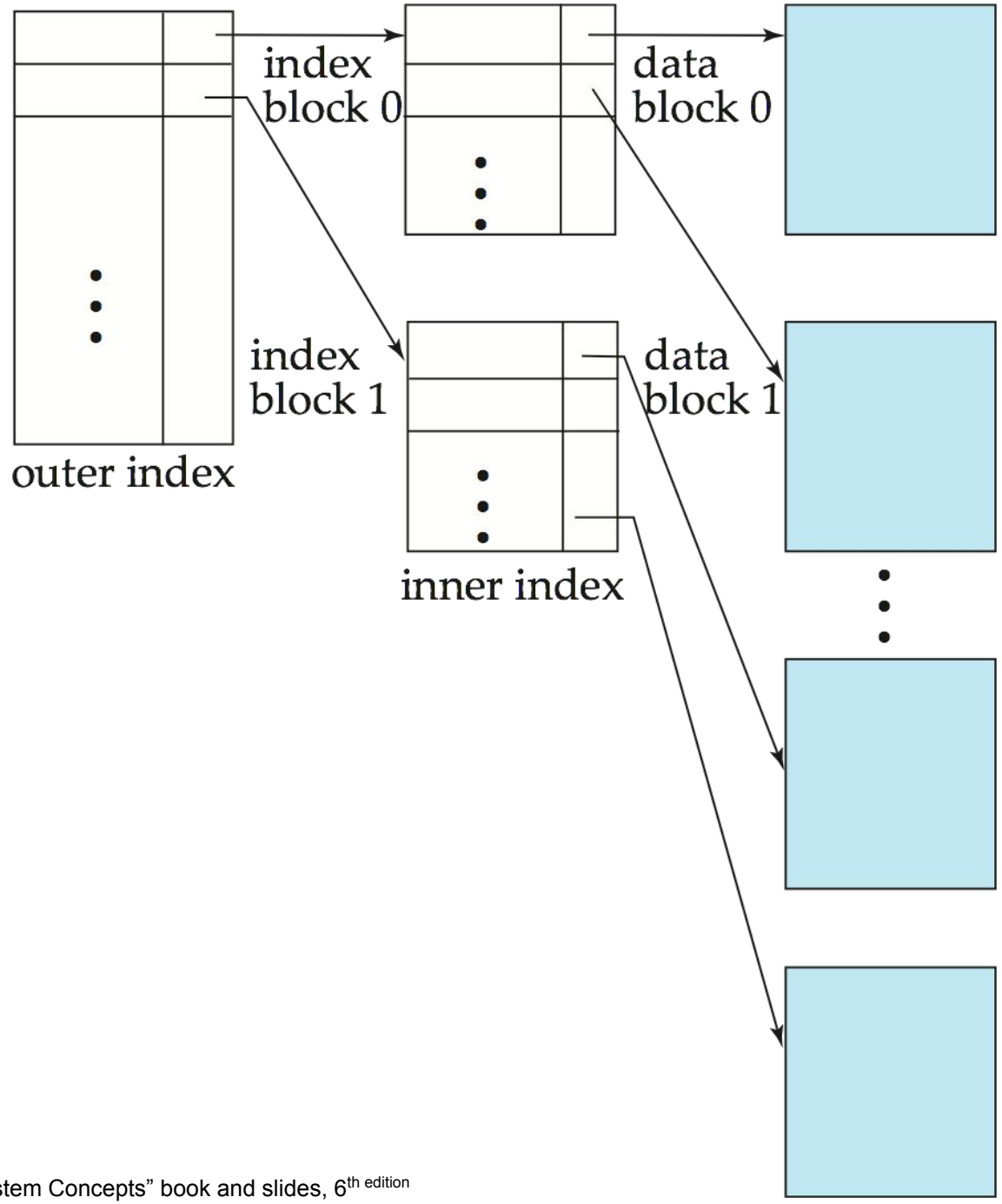
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - 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.

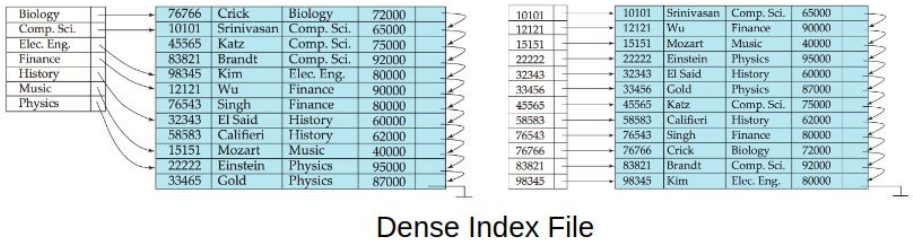
Multilevel Index



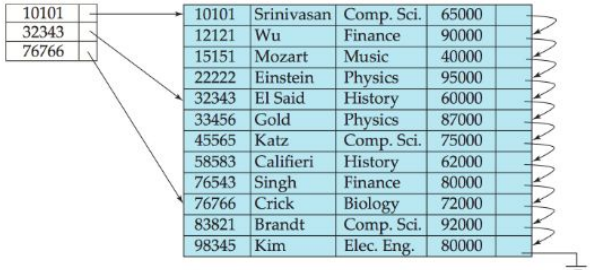
Based on and image from "Database System Concepts" book and slides, 6th edition

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **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.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

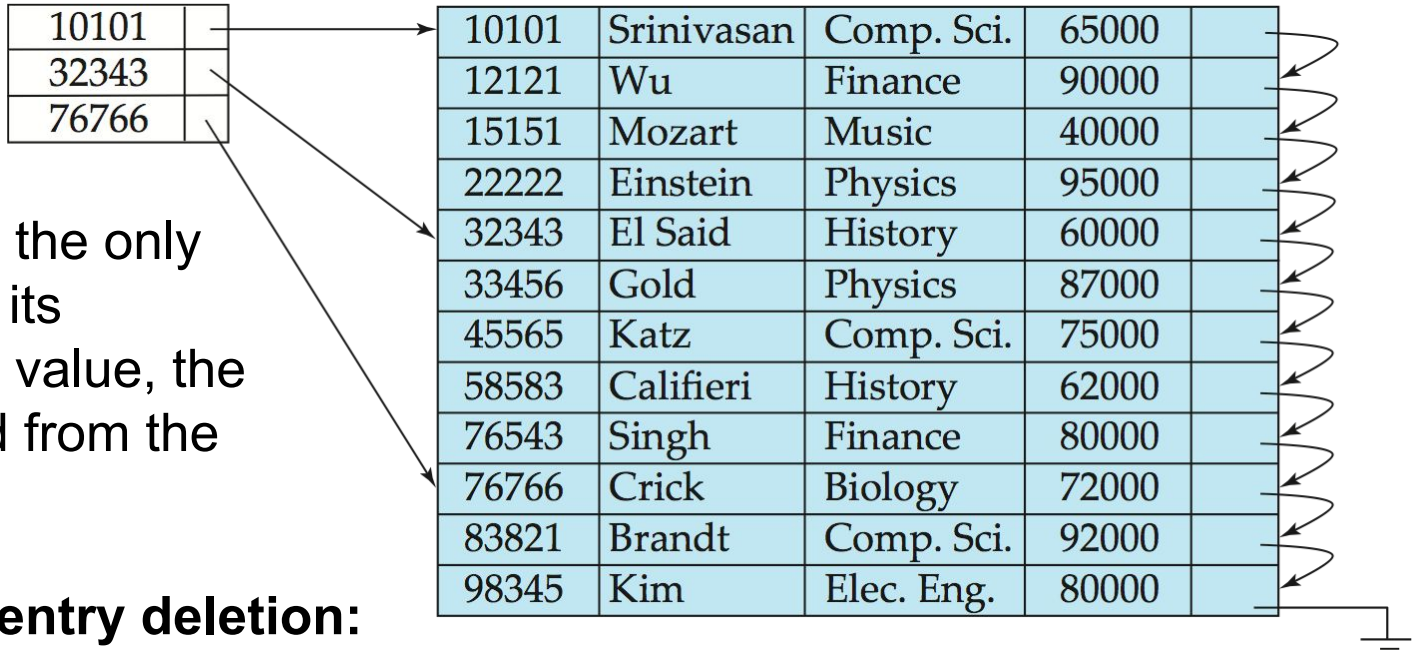


Dense Index File



Sparse Index File

Index Update: Deletion



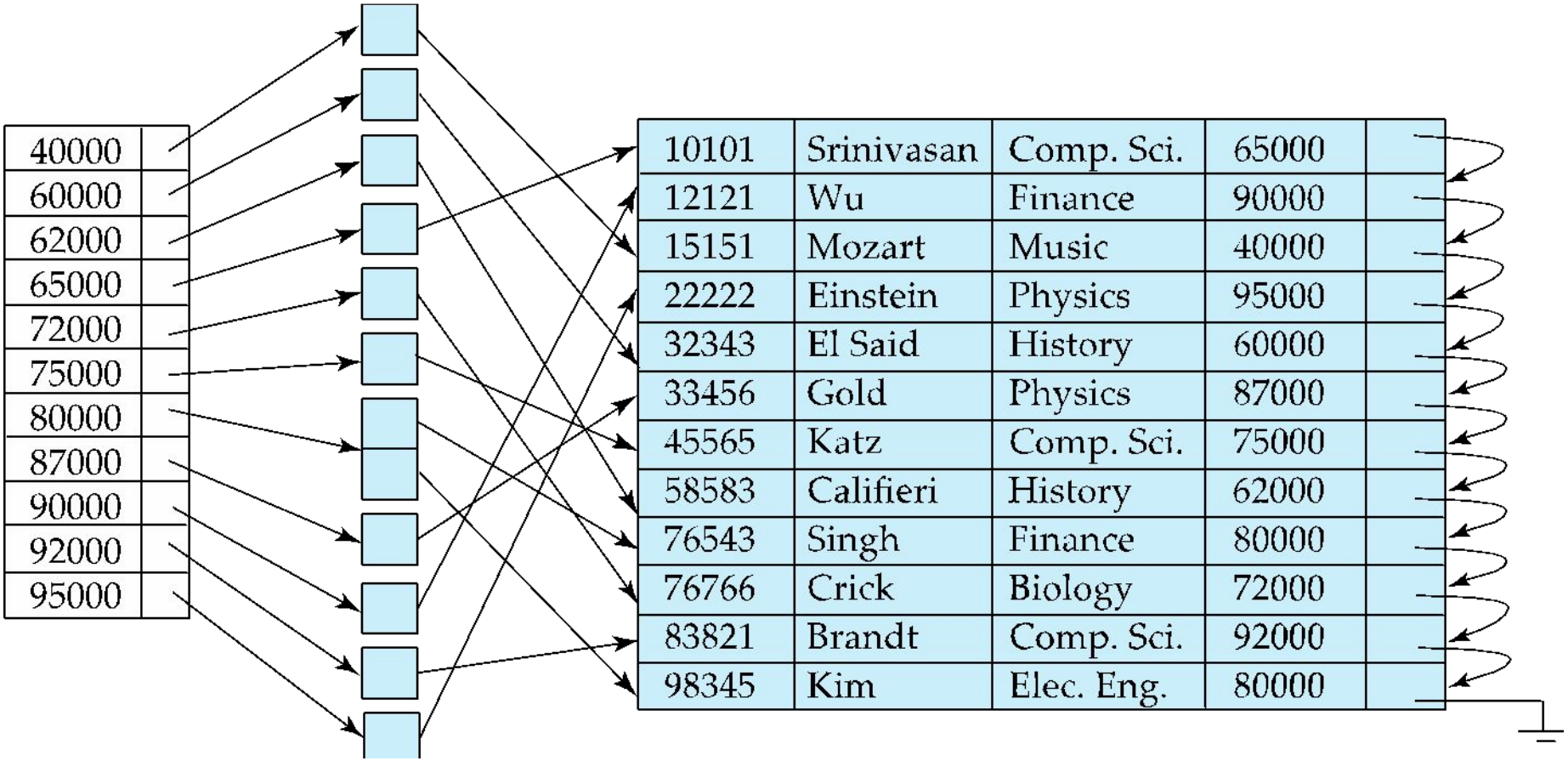
- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

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

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

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